Algorithms on weighted sequences & applications

Liu, Chang

Awarding institution:
King's College London

Algorithms on Weighted Sequences & Applications

Chang Liu

Department of Informatics

School of Natural and Mathematical Sciences

King's College London

This dissertation is submitted for the degree of

Doctor of Philosophy

King's College London May 2019

Declaration

I declare that this doctoral thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text. The following articles were published during my period of research. Certain materials and conceptes from these publications will necessarily be presented within the body of this work.

1. Barton C., Liu C., Pissis S. P., Fast Average-Case Pattern Matching on Weighted Sequences. International Journal of Foundations of Computer Science, 2018. (in press).

2. Barton C., Liu C., Pissis S. P., "On-Line Pattern Matching on Uncertain Sequences and Applications", in *Combinatorial Optimization and Applications: 10th International Conference*, COCOA 2016, Hong Kong, China, December 16–18, 2016, Proceedings, T. H. Chan, M. Li, L. Wang, Eds., Cham: Springer International Publishing, 2016, pp. 547–562.

3. Barton C., Kociumaka T., Liu C., Pissis S.P., Radoszewski J.: Indexing Weighted Sequences: Neat and Efficient. CoRR abs/1704.07625v2 (2017)

4. Charalampopoulos P., Iliopoulos C.S., Liu C., Pissis S.P. (2018) Property Suffix Array with Applications. In: Bender M., Farach-Colton M., Mosteiro M. (eds) LATIN 2018: Theoretical Informatics. LATIN 2018. Lecture Notes in Computer Science, vol 10807. Springer, Cham

5. M. Alzamel, J. Gao, C. S. Iliopoulos, C. Liu, S. P. Pissis, "Efficient Computation of Palindromes in Sequences with Uncertainties", in Engineering Applications of Neural Networks: 18th International Conference, EANN 2017, Athens, Greece, August 25–27, 2017, Proceedings, G. Boracchi et al., Eds., Cham: Springer International Publishing, 2017, pp. 620–629.

The above papers are the only collaborative work included in this thesis. The candidate contributes at least 50% work to each paper mentioned above. The rest of the materials in this thesis is entirely the candidate's contribution.

In this thesis, most of the algorithms are implemented by myself, and published on my GitHub: https://github.com/YagaoLiu/. YagaoLiu is my nickname online.

Chang Liu

May 2019

Acknowledgements

I would like to extend thanks to all the people who contributed to the work presented in this thesis.

Firstly, I would like to express my sincere gratitude to my supervisor Dr. Solon P. Pissis for his continuous support in my study and research. He was also my master supervisor, and it was his invitation that led me to be a PhD student, and finally gave me a wonderful 4-year academic life. During my PhD, I submitted several papers and got opportunities to present our research result in conferences. It is difficult to imagine how I can achieve these without his help.

I would also like to thank my second supervisor Prof. Costas Iliopoulos, not only for many research topics given by him, but also the chances to take part in the organization of the conferences in King's College London. Besides my supervisors, my sincere thanks also goes to Dr. Carl Barton. Nearly half of my work was done with him and I really benefit a lot from his excellent algorithm design and mathematics analysis.

I thank Prof. Roberto Grossi, Dr. Nadia Pisanti, Dr. Jakub Radoszewski, Tomasz Kociumaka and my group mates Panagiotis Charalampopoulos, Ahmad Retha, Fatima Vayani, Mai Alzamel, Jia Gao for the co-operation work during my PhD. I also thank all

other group mates in the Algorithms & Data Analysis group. I really enjoy this warm and interesting group.

Last but not least, I would like to thank my parents for their comprehensive support during my PhD, and also during all of my life.

Abstract

In this thesis, we study problems on a type of uncertain sequences called *weighted sequences*. In weighted sequences, for every position of the sequence and every letter of the alphabet a probability of occurrence of this letter at this position is specified. This type of sequences are commonly used to represent imprecise or uncertain data, for example in molecular biology, where they are known as Position Weight Matrices. Normally in weighted sequences problem a cumulative weight threshold $1/z \in (0,1]$ is given and we only consider the substrings with cumulative probability at least $1/z$, which we call it *valid*. Two main problems on weighted sequences are addressed in this thesis: *pattern matching* and *weighted indexing*. An application of weighted indexing is also presented to solve *palindromic factorization* in weighted sequences.

- In pattern matching problem we are given a pattern of length m, a text of length $n > m$ and a cumulative weight threshold $1/z$. Our aim is to find all valid occurrences of the pattern in the text. We provide two types of average-case algorithms: one is to solve the problem with searching time $o(n)$ and the other one is to solve the problem with searching time $\mathcal{O}(\frac{nz\log m}{m})$. Both algorithms require a specific weighted ratio $\frac{z}{m}$. An

average-case algorithm for approximate pattern matching on weighted sequences is also provided in the thesis.

- In weighted indexing problem we are aiming to build an index containing all the valid substrings of a given weighted sequences of length n. We introduce a type of property string of length nz called *z-estimation* which contains all valid substrings, and provide algorithms to construct weighted suffix tree and weighted suffix array in time and space $\mathcal{O}(nz)$ based on z-estimation.

- In palindromic factorization problem we are given a weighted sequence of length n and a threshold $1/z$. We find all the maximum palindrome in the sequence and give a maximal-palindromic factorization of the given sequence. Our algorithm is based on weighted suffix tree and solve the problem with time and space complexity $\mathcal{O}(nz)$.

Experimental results are given using synthetic or real data with DNA alphabet (A, C, G, T) for all the algorithms to prove our complexity theorem or to present the time and space performance.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Background

1.1.1 Sequencing

Deoxyribonucleic acid, known as DNA, is one of the most famous molecular in biology for its function of being a carrier of genetics information in nearly all living organisms. The structure of DNA, discovered by Francis Crick and James D. Watson in 1953 [64], verified by Rosalind Franklin and Raymond Gosling using X-ray diffraction image of DNA known as Photograph 51 [29], is a double helix of two chains. Each chain is made of nucleotides and each nucleotide is a composition of a deoxyribose, a phosphate group and one of four nucleobases: adenine(A), guanine(G), cytosine(C), or thymine(T). The order in which the nucleotides are ranged in DNA, or generally the order of nucleobases, is called sequence of DNA, and the technique to determine this order is called DNA sequencing.

The earliest DNA sequencing technologies, so called first generation sequencing, including Maxam-Gilbert method [50] and Sanger's method [60], was developed in 1970s. The next generation sequencing (NGS) technologies, developed since 1990s, increased the speed of sequencing and decreased the cost by orders of magnitudes [62]. At the same time, the human genome project was set up in 1990, aiming to determine the whole human DNA sequences, and completed in 2003 with a high-quality sequence of essentially the entire human genome. After the completion of human genome, a new project named 1000 Genomes Project was launched in 2008 and completed in 2015, to sequence and study the genome from at least 1000 people from different ethnic groups, aiming to create a catalogue of human genetic variation. In this 7-year project, totally 2,504 genomes from 26 population were studied and a most comprehensive view of global human variation was provided [16].

1.1.2 Genetic Variation

The genetic variation is the genetic difference between individuals within a population, which leads to individuals' different phenotype, and finally leads to the polymorphism of species. There exists two main forms of variation: single base-pair substitution, known as SNP, and insertion or deletion, also known as "indel".

Example 1.1.1. An example of SNP and indel in DNA sequence:

Reference	AACTACTGAAA **T** AACTACTGAAA
SNP	AACTACTGAAA **C** AACTACTGAAA
Deletion	AACTACTGAAA **-** AACTACTGAAA
Insertion	AACTACTGAAA **TGC** AACTACTGAAA

In order to present the various DNA sequences, the alphabet of DNA has been extended and the new nucleic acid notation was formalised by the International Union of Pure and Applied Chemistry(IUPAC) in 1970. In IUPAC nucleotide code, one letter is used to represent one or several nucleotides. The notation is shown in Table 1.1.

IUPAC code	Base
A	Adenine
C	Cytosine
G	Guanine
T	Thymine
R	A or G
Y	C or T
S	G or C
W	A or T
K	G or T
M	A or C
B	C or G or T
D	A or G or T
H	A or C or T
V	A or C or G
N	A or C or G or T
-	gap

Table 1.1 IUPAC nucleotide code.

In this thesis, we consider a type of uncertain sequence to present various DNA sequences: Weighted Sequence. Weighted sequence is an uncertain sequence in which for every position of the sequence and every letter of the alphabet a probability of occurrence of this letter at this position is specified. It was first introduced by Iliopoulos et al. [38] in 2003, and it is considered to be another form of Position Weight Matrix, which is introduced by Gary Stormo in 1983 [61], used for representing motifs in biological sequences. A great deal of research has been conducted since then on weighted sequences: for pattern matching [21, 13, 44, 57,

7]; for computing various types of regularities [39, 20, 11, 14]; for indexing [37, 17, 12]; and for alignments [8, 26].

Except the using in biological sequences, weighted sequences are also used in a wide range of applications such as data measurements with imprecise sensor measurements and observations that are private and thus sequences of observations may have artificial uncertainty introduced deliberately (see [3] for a survey).

1.2 Structure of This Thesis

In the rest of this chapter, we present some of the definitions used in this thesis. Three main chapters are included in this thesis to present the algorithms on weighted sequences:

1. In Chapter 2, we provide two algorithms to solve pattern matching problem in weighted strings. The first algorithm is an average case algorithm with sublinear searching time, solving weighted pattern matching and weighted text matching with a specific weighted ratio. The second algorithm is an efficient on-line average case algorithm, solving general weighted pattern matching problems as well as approximate weighted matching. We also provide experimental results, using synthetic DNA data and real data to prove our theorems and compare with other algorithms to show our efficiency.

2. In Chapter 3, we provide two algorithms for weighted indexing problem. At the beginning of this chapter we provide a method to construct a property string from a given weighted string. Based on this property string, we provide one algorithm to build weighted suffix tree, and three worse case algorithms and an average case algorithm to

build weighted suffix array. We also provide experimental results using synthetic and real data to prove our theorem and efficiency.

3. In Chapter 4, we provide an algorithm based on weighted suffix tree to solve the palindrome factorisation problem on weighted sequences. Experimental results using synthetic data are provided as well.

At the end of this thesis we give a chapter of conclusion and a discussion on future works.

1.3 Preliminaries

1.3.1 String

An *alphabet* Σ is a finite non-empty set of size σ, whose elements are called *letters*. A *string* on an alphabet Σ is a finite, possibly empty, sequence of elements of Σ. The zero-letter sequence is called the *empty string*, and is denoted by ε. The *length* of a string x is defined as the length of the sequence associated with the string x, and is denoted by $|x|$. We denote by $x[i]$, for all $0 \leq i < |x|$, the letter at index i of x. Each index i, for all $0 \leq i < |x|$, is a *position* in x when $x \neq \varepsilon$. It follows that the i-th letter of x is the letter at position $i-1$ in x. We define that two strings are equal, denoted by $x = y$, if and only if $|x| = |y|$ and $x[i] = y[i]$ for all $0 \leq i < |x|$. The *concatenation* of two strings x and y is the string composed of the letters of x followed by the letters of y; it is denoted by xy. Additionally, the word *string* is only used in algorithm description, stand for sequence of letters, and the word *sequence* is used in most of the real world data, such as DNA sequence or protein sequence.

Example 1.3.1. Given an alphabet $\Sigma = \{\texttt{a},\texttt{b},\texttt{c}\}$, a string $x = \texttt{abac}$ is a string on alphabet Σ of length $|x| = 4$. We have $x[0] = a$, $x[1] = b$, $x[2] = a$, $x[3] = c$. Given a string $y = \texttt{bacbaa}$ on alphabet Σ of length 6, the concatenation of x and y is $xy = \texttt{abacbacbaa}$.

A string x is a *factor* of a string y if there exist two strings u and v, such that $y = uxv$. When $u = \varepsilon$, x is a *prefix* of y; and when $v = \varepsilon$, x is a *suffix* of y. We can also denote by $x[i\,..\,j]$ to be the factor of the string x and $x[i\,..\,j] = x[i]x[i+1]x[i+2]...x[j]$ for integer i and j satisfying $0 \leq i < |x|$, $i \leq j < |x|$. Specifically, $x[i\,..\,j]$ is an empty string if $i = j+1$. We say that a string x is a *proper* factor of string y if x is a factor of y and $x \neq y$.

Example 1.3.2. Given a string $x = \texttt{abaabaaab}$ on $\Sigma = \{\texttt{a},\texttt{b}\}$, string $u = \texttt{abaab}$ is a (proper) prefix of x, and string $v = \texttt{abaaab}$ is a (proper) suffix of x.

Given a non-empty string x and a string y, we say that there exists an *occurrence* of x in y, or more simply, that x *occurs in* y, when x is a factor of y, and if $x = y[i\,..\,i+|x|-1]$, we say that x occurs in y at position i.

Example 1.3.3. Given a string $y = \texttt{abaabaaab}$ of length 8 and $x = \texttt{abaa}$ of length 4, we say x occurs in y at position 0 and 3.

The *lexicographic order*, denoted by $\leq$ or $<$, is an order induced by the letters in alphabet which is denoted by the same symbols. Given two strings x and y on alphabet Σ, $x \leq y$ if and only if x is a prefix of y, or x and y can be decomposed into $x = u\texttt{a}v$ and $y = u\texttt{b}w$, where u, v and w are strings on Σ and a, b are letters which hold $\texttt{a},\texttt{b} \in \Sigma$ and $\texttt{a} < \texttt{b}$.

Example 1.3.4. Given three strings on $\Sigma = \{\texttt{a},\texttt{b}\}$: $u = \texttt{abaa}$ of length 4, $v = \texttt{abbab}$ of length 5 and $w = \texttt{abaaab}$, the lexicographic order is $u < w < v$ if it holds $\texttt{a} < \texttt{b}$.

1.3.2 Weighted String

A *weighted string* X of length $|X| = n$ on an alphabet Σ is a finite sequence of n sets of different letters in the alphabet. Every $X[i]$, for all $0 \leq i < n$, is a set of ordered pairs $(s_j, \pi_i(s_j))$, where $s_j \in \Sigma$ and $\pi_i(s_j)$ is called the *occurrence probability* of letterss_j, which denotes the probability of having letter s_j at position i. Formally, $X[i] = \{(s_j, \pi_i(s_j)) | s_j \neq s_\ell \text{ for } j \neq \ell, \text{ and } \sum_j \pi_i(s_j) = 1\}$. Note that for clarity we use upper case letters for weighted strings, e.g. X, and lower case letters, e.g. x, for standard strings.

A weighted string can be presented as a $n \times \sigma$ matrix. An example of a weighted string on alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ is shown below.

Example 1.3.5. An example of a weighted string on alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$

i	0	1	2	3	4	5	6
$\pi_i(\mathtt{a})$	1	0.5	0.7	0.2	1	0	0.75
$\pi_i(\mathtt{b})$	0	0.5	0.3	0.8	0	1	0.25

Table 1.2 A weighted string X of length 7 on $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.

For succinctness of presentation, if $\pi_i(s_j) = 1$ the set of pairs is denoted only by the letter s_j; otherwise it is denoted by $[(s_{j_1}, \pi_i(s_{j_1})), \ldots, (s_{j_k}, \pi_i(s_{j_k}))]$. Thus the weighted string in Example 1.3.5 can be denoted by:

$X =$ `a[(a,0.5),(b,0.5)][(a,0.7),(b,0.3)][(a,0.2),(b,0.8)]ab[(a,0.75),(b,0.25)]`

The *cumulative occurrence probability* is a product of occurrence probability of a specific letter at each position. Given a string u of length m, the cumulative occurrence probability of

u at position i in a weighted string x is denoted by:

$$\prod_{j=0}^{m-1} \pi_{i+j}(u[j]) = \pi_i(u[0]) \times \pi_{i+1}(u[1]) \times .. \times \pi_{i+m-1}(u[m-1])$$

A letter s_j *occurs* at position i in a weighted string X if and only if the occurrence probability of letter s_j at position i, $\pi_i(s_j)$, is greater than 0. A string u of length m is a *factor* of a weighted string if and only if it occurs at starting position i with cumulative occurrence probability greater than 0.

A *cumulative weight threshold*, or threshold for short, denoted by $1/z$, is a positive number between 0 and 1. Given a cumulative weight threshold $1/z \in (0,1]$, we say that factor u is *valid*, or equivalently that factor u has a valid occurrence, if it occurs at starting position i and has $\prod_{j=0}^{m-1} \pi_{i+j}(u[j]) \geq 1/z$. Similarly, we say that letter s_j at position i is valid if $\pi_i(s_j) \geq 1/z$.

Example 1.3.6. Given the weighted string X in Example 1.3.5 and a cumulative weight threshold $1/z = 0.25$. We say a string $u = \texttt{aaba}$ is valid at position 1 since

$$\prod_{j=0}^{3} \pi_{j+1}(u[j]) = 0.5 \times 0.7 \times 0.8 \times 1 = 0.28 > 1/z = 0.25$$

We say a string $v = \texttt{bbba}$ is not valid, or invalid, at position 1 since

$$\prod_{j=0}^{3} \pi_{j+1}(v[j]) = 0.5 \times 0.3 \times 0.8 \times 1 = 0.12 < 1/z = 0.25$$

We say that a position in a weighted string is a *uncertain position*, if there exist at least two letters at this position, and each of these letters has occurrence probability greater or equal that $1/z$. By δ, we denote the *uncertain positions percentage* of a weighted string.

Example 1.3.7. Given a weighted string X of length n, assuming there exist k uncertain positions in X, the uncertain positions percentage of X is $\delta = k/n$.

1.3.3 Pattern Matching

Pattern matching is a fundamental problem in computer science and bioinformatics, aiming to search and locate some specific sequences of some patterns in a given raw data or a sequence of token, and has many real world applications including computer virus detection, genome assembly, database search.

In string processing algorithms, there exist two kinds of matching: exact matching and approximate matching. Exact matching is simple: two strings u and v are exact matched if $u = v$. Therefore we define exact pattern matching problem as follow:

Given a string x of length m called *pattern* and a string y of length $n \geq m$ called *text*, the pattern marching problem is to find all the positions i in y such that $x = y[i..i+n-1]$.

Example 1.3.8. Given a pattern $x =$ `abbaa` of length 5 and a text $y =$ `aabbaabbaa` of length 10. The pattern matching to find x in y will report two occurrences at starting position 1 and 5.

Unlike the exact pattern matching, in approximate pattern matching problem the pattern may not exactly match with a factor of the text. Instead, there may exist "errors" between the

pattern and a factor in the text. The errors include insertion, deletion and substitution, and in this thesis we only concern the case of substitution.

- Insertion: `ab-ba` $\rightarrow$ `ababa`.
- Deletion: `ababa` $\rightarrow$ `ab-ba`.
- Substitution: `ababa` $\rightarrow$ `abbba`

We say that two strings u and v both of length m match with k *mismatches* if we can transform u to v with k substitutions. Therefore, we can define approximate pattern matching problem as follow:

Given a string x of length m, a text y of length $n \geq m$ and a positive integer $k < m$ the approximate matching problem is to find all position i in y such that x and $y[i..i+m-1]$ match with no more than k mismatches.

Further definition of pattern matching in weighted string will be presented in Chapter 2.

1.3.4 String Indexing

In this subsection we introduce two powerful index data structures in string processing algorithms: suffix tree and suffix array.

The *suffix tree* T of a non-empty string s of length n is a compact trie representing all suffixes of s. The nodes of the trie which become nodes of the suffix tree (i.e., branching nodes, terminal nodes, and the root) are called *explicit* nodes, while the other nodes are called *implicit*. The edges out-going from a node are labelled with their first letters and can be stored, e.g., in a list.

Each edge of the suffix tree can be viewed as an upward maximal path of implicit nodes starting with an explicit node. Moreover, each node belongs to a unique path of that kind. Then, each node of the trie can be represented in the suffix tree by the edge it belongs to and an index within the corresponding path. We use $\mathsf{L}(v)$ to denote the *path-label* of a node v, i.e., the concatenation of the edge labels along the path from the root to v. The terminal node corresponding to suffix $S[i..n]$ is marked with the index i. Each string p occurring in s is uniquely represented by either an explicit or an implicit node of T, called the *locus* of p. The *suffix link* of a node v with path-label $\mathsf{L}(v) = cp$ is a pointer to the node path-labelled p, where $c \in \Sigma$ is a single letter and p is a string. The suffix link of every non-root explicit v leads to an explicit node of T.

Fact 1.3.1 ([27])**.** *The suffix tree of a string of length n even over an integer alphabet* Σ*, i.e.,* $\Sigma \subseteq \{1,\ldots,n^{\mathcal{O}(1)}\}$ *can be constructed in time and space* $\mathcal{O}(n)$*. Checking whether a string x of length m occurs in y can be performed in time* $\mathcal{O}(m)$*.*

Example 1.3.9. Given a string $x = \texttt{abbabc}$, the suffix tree of x is plotted in Fig 1.1.

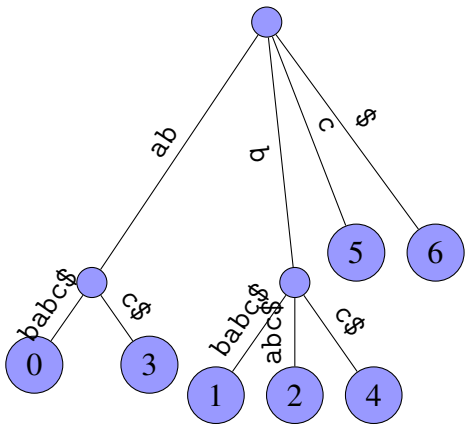


Fig. 1.1 Suffix tree of string $x = \texttt{abbabc}$

We denote by SA the *suffix array* of a non-empty string x of length n. SA is an integer array of size n storing the starting positions of all (lexicographically) sorted non-empty suffixes of x, i.e. for all $1 \leq r < n$ we have $x[\mathsf{SA}[r-1]..n-1] < x[\mathsf{SA}[r]..n-1]$ [49]. Let $\mathsf{lcp}(r,s)$ denote the length of the longest common prefix between $x[\mathsf{SA}[r]..n-1]$ and $x[\mathsf{SA}[s]..n-1]$ for all positions r, s on SA, and 0 otherwise. We denote by LCP the *longest common prefix* array of y defined by $\mathsf{LCP}[r] = \mathsf{lcp}(r-1,r)$ for all $1 \leq r < n$, and $\mathsf{LCP}[0] = 0$. The inverse iSA of the array SA is defined by $\mathsf{iSA}[\mathsf{SA}[r]] = r$, for all $0 \leq r < n$. It is known that SA [56], iSA, and LCP [43] of a string of length n, over an integer alphabet, can be computed in time and space $\mathcal{O}(n)$. It is then known that a range minimum query (RMQ) data structure over the LCP array, that can be constructed in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space [15], can answer lcp queries in $\mathcal{O}(1)$ time per query by returning the index of a minimal value in the respective range of the SA.

Example 1.3.10. Given a string $x =$ `abbabc`, the suffix array of x is plotted in Table 1.3.

i	suffix
0	$
1	`abbabc$`
2	`abc$`
3	`babc$`
4	`bbabc$`
5	`bc$`
6	`c$`

Table 1.3 Suffix array of string $x =$ `abbabc`

Chapter 2

Pattern Matching in Weighted Sequences

2.1 Pattern Matching on Weighted Sequences

In this chapter, we apply pattern matching problems on weighted sequences. Before we go directly to the main problem, we need to define *matching* in weighted string. There exist two cases of matching in weighted string: (i) a string matches with a weighted string, and (ii) two weighted strings match each other. Assuming we are given a cumulative weight threshold $1/z$, we say that a string u and a weighted string X match, denoted by $u =_z X$, if and only if $|u| = |X|$ and $\prod_{i=0}^{|u|-1} \pi_X(u[i]) \geq 1/z$. Given a string u and a weighted string X, both of length m, and given a positive integer $k < m$, we say that u and X match with *k-mismatches*, denoted by $u =_{z,k} X$, if there exists a string u' which is created by replacing at most k letters from u and $u' =_z X$. We say that two weighted string X and Y match, denoted by $X =_z Y$, if and only if $|X| = |Y|$ and there exists a string u such that u matches with both X and Y.

Example 2.1.1. Given a cumulative weight threshold $1/z = 0.25$ and a weighted string of length 6:

$$X = \texttt{a[(a,0.5),(b,0.5)][(a,0.7),(b,0.3)]ab[(a,0.8),(b,0.2)]}$$

Given a string $u = \texttt{abaaba}$, we say that $u =_z X$ since $\prod_{i=0}^{5} \pi_i(u[i]) = 0.28 \geq 1/z = 0.25$.

Given a string $v = \texttt{bbaabb}$, we say that $v =_{z,2} X$ since we can replace the letter at position 0 and 5 to construct a string $v' = \texttt{abaaba}$ and $v' =_z X$.

Given a weighted string $Y = \texttt{[(a,0.75),(b,0.25)]baab[(a,0.6),(b,0.4)]}$, we say that $Y =_z X$ since there exists a string $w = \texttt{abaaba}$ and $w =_z X$, $w =_z Y$.

With this definition of matching, we are able to describe the pattern matching problems on weighted sequences:

WEIGHTEDPATTERNMATCHING
Input: a weighted string X of length m, a string y of length $n > m$, and a cumulative weight threshold $1/z \in (0,1]$
Output: all positions i of y such that $y[i..i+m-1] =_z X$

WEIGHTEDTEXTMATCHING
Input: a string x of length m, a weighted string Y of length $n > m$, and a cumulative weight threshold $1/z \in (0,1]$
Output: all positions i of Y such that $x =_z Y[i..i+m-1]$.

GENERALWEIGHTEDPATTERNMATCHING

Input: a weighted string X of length m, a weighted string Y of length $n > m$, and a cumulative weight threshold $1/z \in (0,1]$

Output: all positions i of Y such that $X =_z Y[i..i+m-1]$

APPROXWEIGHTEDTEXTMATCHING

Input: a string x of length m, a weighted string Y of length $n > m$, a non-negative integer $k < m$, and a cumulative weight threshold $1/z \in (0,1]$

Output: all positions i of y such that $x =_{z,k} Y[i..i+m-1]$

Previous Results In what follows, we assume that $\sigma = \mathcal{O}(1)$ since the most commonly studied alphabet is DNA alphabet $\Sigma = \{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}$. In the off-line setting an $\mathcal{O}(n\log m)$-time solution for these problems based on Fast Fourier Transform was proposed in [21]. Another $\mathcal{O}(nz^2\log z)$-time algorithm for solving both problems was presented in [13]. Kociumaka et al. presented an $\mathcal{O}(n\log z)$-time algorithm for these problems [44].

2.2 Average-Case $o(n)$-time Pattern Matching Algorithms on Weighted Sequences

2.2.1 Our Contribution

In this section, we present two new on-line algorithms: one to solve problem WEIGHTED-PATTERNMATCHING and another one to solve problem WEIGHTEDTEXTMATCHING. Both algorithms can achieve average-case *sublinear* search time in the size of the text, and work in *linear* preprocessing time and space with respect to the size of the pattern. Essentially, we show that they achieve these average-case search times depending on the number of positions required, in every matching weighted segment of length m, to have a letter occurring with probability at least $1-1/z$. Moreover, we present extensive experimental results, using both real and synthetic data, showing that our implementations are indeed practical: they are between one and two orders of magnitude faster than the best worst-case approaches and competitive to the existing average-case approaches.

2.2.2 Properties and Auxiliary Data Structures

Given a weighted string X of length m, we perform a colouring stage on X, which assigns a colour to every position in X according to the following schemes:

- mark position i *black* (**b**), if *none* of the occurring letters at position i has probability of occurrence greater than $1-1/z$.

- mark position *i grey* (**g**), if *one* of the occurring letters at position *i* has probability of occurrence greater than $1-1/z$ and less than 1.

- mark position *i white* (**w**), if *one* of the occurring letters at position *i* has probability of occurrence 1.

Notice that if $z \geq 2$, then at every white and grey position there is only one valid letter since only one letter can have probability of occurrence greater than $1-1/z \geq 1/2$, whereas in a black position there maybe several valid letters. However, if $z < 2$ there are no valid letters with probability of occurrence at most $1-1/z$ since $1-1/z < 1/z$ and therefore no black positions. Therefore for the rest of this paper we assume $z \geq 2$. The colouring stage can be trivially performed in time $\mathcal{O}(\sigma m) = \mathcal{O}(m)$.

Lemma 2.2.1 ([7])**.** *Given a weighted string x and a cumulative weight threshold* $1/z \in (0,1]$*, any valid factor of x contains at most* $\ell = \lfloor \log z / \log(\frac{z}{z-1}) \rfloor$ *black positions.*

Proof. Any letter at a black position of x has probability of occurrence less than or equal to $1-1/z$. For any valid factor of x, it thus holds (in the worst case) that $(1-1/z)^{\ell} \geq 1/z$. Taking the logarithm at both sides yields the lemma. □

The second key idea of the designed algorithms comes from the following simple fact. This idea is used in many *other* pattern matching problems on strings [54].

Lemma 2.2.2. *Given a weighted string X and a cumulative weight threshold* $1/z \in (0,1]$*, if* $\ell < m$*, then there exists a consecutive sequence of positions of length at least* $\lfloor \frac{m}{\ell+1} \rfloor$ *of X consisting of only white and grey positions.*

Proof. Immediate from the pigeonhole principle. □

We can also preprocess a weighted string X of length m to compute a matrix $\mathsf{A}[0\ldots\ell'-1,0\ldots\sigma-1]$, such that for each black position i, $0\leq i<\ell'$, and each letter $a\in\Sigma$, we have $\mathsf{A}[i,\alpha]=1$ if α occurs at the ith black position of X and 0 otherwise. After such an $\mathcal{O}(m)$-time preprocessing, we can check in constant time whether a letter in a black position of X matches a letter from another string or not. With matrix A at hand, we can proceed with a fast verification step of the precomputed candidate occurrences using Lemma 2.2.3 (see below).

Given two strings u and v in the standard setting, we say that the probability that $u[i]=v[i]$, for some position i on u and v, is given by $1/\sigma$ assuming uniform distribution of letters of the alphabet per position. This randomness model cannot be applied on weighted strings, where a subset of the alphabet occurs at every position of the string. For a given position, we rather assume a uniform distribution of all possible non-empty subsets of the alphabet, such that each letter of the subset has probability of occurrence greater than 0; i.e., such that the letter occurs. Under this model we can obtain the following lemma.

Lemma 2.2.3. *Given a string u and a weighted string V, the expected length of the longest valid prefix of V that is also a prefix of u is bounded by six.*

We start with a few definitions to reduce this problem to another one before giving a proof. An *indeterminate string* X of length m on an alphabet Σ is a finite sequence of m sets, such that $X[i]\subseteq\Sigma$, $X[i]\neq\emptyset$, for all $0\leq i<m$. If $|X[i]|=1$, that is, $X[i]$ represents a single letter of Σ, we say that $X[i]$ is a *solid* letter. We say that two indeterminate strings X and Y *match*, denoted by $X\approx Y$, if $|X|=|Y|$ and for each $i=0,\ldots,|X|-1$, we have $X[i]\cap Y[i]\neq\emptyset$.

Proof. [of Lemma 2.2.3] We view the weighted string V as indeterminate string V' of length $|V|$ such that $a \in V'[i]$ if and only if $(a, \pi(a)) \in V[i]$ and $\pi(a) > 0$. Since we completely ignore letter probabilities and thereby the validity of factors—all factors are now valid—it suffices to show that the expected number $s > 0$ of positions such that $U[0..s-1] \approx V'[0..s-1]$ and $U[s] \notin V'[s]$ is bounded by six.

We consider the comparison of u and v' from left to right. We have that $\{U[i]\} \cap V'[i] \neq \emptyset$ occurs in the following cases:

- $V'[i]$ is solid and $\{U[i]\} = V'[i]$
- $V'[i]$ is not solid and $U[i] \in V'[i]$.

Thus the total number of positive comparisons is

$$\sigma \sum_{j=1}^{\sigma} \frac{j\binom{\sigma}{j}}{\sigma} = \sum_{j=1}^{\sigma} j\binom{\sigma}{j} = \sigma 2^{\sigma-1}.$$

The total number of any case is $\sigma \sum_{j=1}^{\sigma} \binom{\sigma}{j} = \sigma(2^{\sigma}-1)$. Therefore the probability r of $\{U[i]\} \cap V'[i] \neq \emptyset$ is

$$r = \frac{2^{\sigma-1}}{2^{\sigma}-1} \leq 2/3, \text{ for } \sigma > 1.$$

Thus the expected number $s > 0$ of positions such that $U[0..s-1] \approx V'[0..s-1]$ and $U[s] \notin V'[s]$ can be described by the summation of infinite terms

$$s = r + 2r^2 + \cdots = \sum_{k=1}^{\infty} kr^k,$$

which is bounded by $r/(1-r)^2 \leq 6$, for $r \leq 2/3$. This concludes the proof. □

Lemma 2.2.4. *Let $z \geq 2$. Then $\ell \leq z \log z$.*

Proof. By Lemma 2.2.1 we know that $\ell = \lfloor \frac{\log z}{\log(\frac{z}{z-1})} \rfloor$. For $z > 1$, we must show that:

$$\ell = \left\lfloor \frac{\log z}{\log(\frac{z}{z-1})} \right\rfloor = \left\lfloor \frac{\log z}{\log(z) - \log(z-1)} \right\rfloor \leq z \log z.$$

Or equivalently that:

$$\frac{\log z(z \log z - z \log(z-1) - 1)}{\log z - \log(z-1)} > 0.$$

Clearly the above is true if and only if: $z \log z - z \log(z-1) - 1 > 0$. There is a discontinuity at $z = 1$; after this it is *always* positive and the following holds:

$$\lim_{z \to \infty} z \log z - z \log(z-1) - 1 = 0.$$

□

2.2.3 Algorithms

WEIGHTEDPATTERNMATCHING

We are now in a position to present Algorithm WPM to solve problem WEIGHTEDPATTERNMATCHING. In this problem, we are given a weighted string X of length m, a string y of length $n > m$, and a cumulative weight threshold $1/z \in (0, 1]$, and we need to find all positions i in y where a valid factor of length m of X occurs.

Algorithm *WPM*$(X, m, y, n, 1/z, \Sigma)$

Perform the colouring stage on X;

Find the number ℓ' of black positions in X;

$\sigma \leftarrow |\Sigma|$;

Compute $\mathsf{A}[0 \dots \ell' - 1, 0 \dots \sigma - 1]$ of X;

if $\ell' < m$ **then**

Find the longest factor f in X with no black positions;

else

return `FAIL`;

Search for f in y on-line;

foreach *occurrence of* f *in* y **do**

Check if f is extensible to the left using A;

Check if f is extensible to the right using A;

if *the length of extension is at least* m **then**

Verify the validity of the factor of X and report the position;

Theorem 2.2.5. *Algorithm WPM correctly solves problem* WEIGHTEDPATTERNMATCHING *in an on-line manner, achieving average-case search time* $o(n)$*, if*

$$\frac{z}{m} < \min\left\{\frac{1}{2\log z + \frac{1}{z}}, \frac{\log \sigma}{\log z(\log m + \log\log \sigma)}\right\}.$$

Algorithm WPM requires preprocessing time and space $\mathcal{O}(m)$.

Proof. Let $\ell < m$. In this case, by Lemma 2.2.2 and the correctness of the average-case time-optimal searching algorithm [65, 23], all positions i of y where a valid factor of length m of X occurs must be naïvely verified in the **for** loop of the algorithm; therefore the algorithm is correct.

The colouring stage on string X can be trivially performed in time $\mathcal{O}(\sigma m) = \mathcal{O}(m)$. The preprocessing time and space cost for array A of x is $\mathcal{O}(\sigma m) = \mathcal{O}(m)$. Assuming $\ell < m$, by Lemma 2.2.2, the minimum factor length in X with no black positions is at least $\lfloor \frac{m}{\ell+1} \rfloor$. This factor f is viewed as a standard string obtained by choosing in all grey positions the most probable letter. Searching for f in y can be performed in average-case time $\mathcal{O}(\frac{n\log(m/\ell)}{m/\ell})$ [65]. The preprocessing time and space for searching is $\mathcal{O}(m)$. The number of expected occurrences is no more than $n/\sigma^{\lfloor m/(\ell+1)\rfloor}$.

Let us denote the average cost of verification per occurrence by $\mathsf{VER}(m,z)$. Algorithm WPM achieves average-case search time $\mathcal{O}(\frac{n\log(m/\ell)}{m/\ell}) = o(n)$ if

$$\frac{\mathsf{VER}(m,z)n}{\sigma^{\lfloor m/(\ell+1)\rfloor}} \leq c\frac{n(\ell+1)\log_\sigma \frac{m}{\ell+1}}{m}$$

for some fixed constant c. That is, the total average-case verification cost is no more than the average-case searching cost. We take σ-based logarithms to obtain

$$\log_\sigma \mathsf{VER}(m,z) + \log_\sigma m - \log_\sigma c - \log_\sigma(\ell+1) - \log_\sigma \log_\sigma \frac{m}{\ell+1} \leq m/(\ell+1).$$

By Lemma 2.2.3 and using array A of X, we know it is possible to pick $c = \mathsf{VER}(m,z)$ to obtain a maximum value for ℓ, that is

$$\log_\sigma \frac{m}{(\ell+1)\log_\sigma \frac{m}{\ell+1}} \leq m/(\ell+1)$$

which gives

$$\ell < m\left(\frac{1}{\log_\sigma \frac{m}{(\ell+1)\log_\sigma \frac{m}{\ell+1}}}\right).$$

Therefore we get the following second condition on ℓ, simplified slightly for comprehension,

$$\ell < m\left(\frac{\log\sigma}{\log m - \log(\ell+1) + \log\log\sigma - \log(\log m - \log(\ell+1))}\right).$$

This gives a third condition on ℓ, tighter with respect to $\ell < m$, namely, $\ell < \frac{m-1}{2}$. By Lemma 2.2.4, we have that $\ell \leq z\log z$, and so from the above we obtain

$$\frac{z}{m} < \min\left\{\frac{1}{2\log z + \frac{1}{z}}, \frac{\log\sigma}{\log z(\log m + \log\log\sigma)}\right\}.$$

□

Example 2.2.1. Running example for algorithm WPM

Given a cumulative weight threshold $1/z = 0.1$, a weighted string X of length $m = 8$ as pattern:

i	0	1	2	3	4	5	6	7
$\pi_i(\mathtt{a})$	1	0.25	0	0.9	0	0.25	1	0
$\pi_i(\mathtt{c})$	0	0.25	0	0	0	0	0	1
$\pi_i(\mathtt{g})$	0	0.25	0	0.05	1	0	0	0
$\pi_i(\mathtt{t})$	0	0.25	1	0.05	0	0.75	0	0

and a string y of length $n = 20$ as text on alphabet $\Sigma = \{\mathtt{a},\mathtt{c},\mathtt{g},\mathtt{t}\}$:

$$y = \mathtt{tatagcaatagtactagaac}$$

First we perform the colouring stage on X:

$$colour(X) = \mathbf{wbwgwbww}$$

We can find $\ell' = 2$, the longest factor $f = \mathtt{tag}$ from position 2 to 4 and compute A:

	a	c	g	t
0	1	1	1	1
1	1	0	0	1

We search f in y and find three occurrences at position 2,8 and 14. The occurrence at position 2 is not extensible because of mismatch: we have $y[0\,..\,7] = \mathtt{tatagcaa}$ where $y[0] = \mathtt{t}$ and $X[0]$ is a white position with the letter a. . The occurrences at position 8 and 14 are able to be extended to length 8, and we have $y[6\,..\,13] = \mathtt{aatagtac}$ and $y[12\,..\,19] = \mathtt{actagaac}$. We

verify the cumulative occurrence probability:

$$\prod_{i=0}^{7} \pi_i(y[i+6]) = 0.25 \times 0.9 \times 0.75 = 0.16875 > 1/z = 0.1$$

$$\prod_{i=0}^{7} \pi_i(y[i+12]) = 0.25 \times 0.9 \times 0.25 = 0.05625 < 1/z = 0.1$$

Therefore we only report position 6 as the starting position of occurrence of X in y.

WEIGHTEDTEXTMATCHING

We next present Algorithm WTM to solve problem WEIGHTEDTEXTMATCHING. In this problem, we are given a string x of length m, a weighted string Y of length $n > m$, and a cumulative weight threshold $1/z \in (0,1]$, and we need to find all positions i of Y where a valid factor v of length m in Y occurs and $v = x$.

For the searching stage of Algorithm WTM, we view the weighted string Y as the (standard) string y' according to the following scheme:

- if position i is white, then $y'[i] = \alpha$, where $(\alpha, \pi(\alpha)) \in Y[i]$, $\alpha \in \Sigma$, and $\pi(\alpha) = 1$.
- if position i is grey, then $y'[i] = \alpha$, where $(\alpha, \pi(\alpha)) \in Y[i]$, $\alpha \in \Sigma$, and $\pi(\alpha) > 1 - 1/z$.
- if position i is black, then $y'[i] = \lambda$, where $\lambda \notin \Sigma$.

Intuitively, while searching, we assign for every black position of Y to y' a letter λ that is not in Σ. This in turn implies that writing a position on string y' requires time $\mathcal{O}(\sigma) = \mathcal{O}(1)$. In [24], it was shown that searching for a set of patterns of total length M in a text of length n requires average-case search time $\mathcal{O}(\frac{n \log m}{m})$ if M is polynomial with respect to the length

m of the shortest pattern in the set. In this case we can do searching for a set of patterns in y' in average-case time $\mathcal{O}(\frac{\sigma n \log m}{m}) = \mathcal{O}(\frac{n \log m}{m})$. The additional factor σ is due to the cost of writing and, subsequently, reading a letter of string y'. Notice that string y' is implicit, we never actually construct it; and we never perform a colouring stage on y as these would require time $\mathcal{O}(\sigma n) = \mathcal{O}(n)$.

Algorithm $\mathit{WTM}(x, m, Y, n, 1/z, \Sigma)$

$\ell \leftarrow \lfloor \log z / \log(\frac{z}{z-1}) \rfloor$;

if $\ell < m$ **then**

Partition x in $\ell + 1$ non-overlapping fragments $f_0, f_1, \ldots, f_\ell$;

Each fragment is of length at most $\lceil \frac{m}{\ell+1} \rceil$ and at least $\lfloor \frac{m}{\ell+1} \rfloor$;

else

return `FAIL`;

Search for $f_0, f_1, \ldots, f_\ell$ by considering string y' on-line;

foreach *occurrence of* $f \in \{f_0, f_1, \ldots, f_\ell\}$ *in* y' **do**

Check if f is extensible to the left naïvely;

Check if f is extensible to the right naïvely;

if *the length of extension is at least* m **then**

Verify the validity of the factor of Y and report the position;

Theorem 2.2.6. *Algorithm WTM correctly solves problem* WEIGHTEDTEXTMATCHING *in an on-line manner, achieving average-case search time $o(n)$, if*

$$\frac{z}{m} < \min\left\{\frac{1}{2\log z + \frac{1}{z}}, \frac{\log\sigma}{\log z(\log m + \log\log\sigma)}\right\},$$

Algorithm WTM requires preprocessing time and space $\mathcal{O}(m)$.

Proof. Let $\ell < m$. In this case, by Lemma 2.2.2 and the correctness of the average-case time-optimal searching algorithm [24], all positions i of Y where a valid factor v of Y occurs, such that $x = v$, must be naïvely verified in the **for** loop of the algorithm; therefore the algorithm is correct.

Assuming $\ell < m$, by Lemma 2.2.2, the minimum factor length with no black positions in any factor of length m of Y is at least $\lfloor \frac{m}{\ell+1} \rfloor$. The searching stage for the $\ell+1$ fragments of x (of length at most $\lceil \frac{m}{\ell+1} \rceil$ and at least $\lfloor \frac{m}{\ell+1} \rfloor$) in string y' can be performed in average-case search time $\mathcal{O}(\frac{\sigma n \log(m/\ell)}{m/\ell}) = \mathcal{O}(\frac{n\log(m/\ell)}{m/\ell})$ (see [24] and the analysis above). The preprocessing time and space for searching is $\mathcal{O}(m)$. The number of expected occurrences is no more than $(\ell+1)n/\sigma^{\lfloor m/(\ell+1)\rfloor}$.

Let us denote the average cost of verification per occurrence by $\sigma \cdot \mathsf{VER}(m,z)$. The additional factor σ is due to the cost of reading a letter of string y naïvely. Algorithm WTM achieves average-case search time $\mathcal{O}(\frac{\sigma n \log(m/\ell)}{m/\ell}) = o(\sigma n) = o(n)$ if

$$\frac{(\ell+1)\sigma \cdot \mathsf{VER}(m,z)n}{\sigma^{\lfloor m/(\ell+1)\rfloor}} \leq c\frac{\sigma n(\ell+1)\log_\sigma \frac{m}{\ell+1}}{m}$$

for some fixed constant c. That is, the total average-case verification cost is no more than the average-case searching cost. We take σ-based logarithms to obtain

$$\log_\sigma \mathsf{VER}(m,z) + \log_\sigma m - \log_\sigma c - \log_\sigma \log_\sigma \frac{m}{\ell+1} \leq m/(\ell+1).$$

By Lemma 2.2.3, we know it is possible to pick $c = \mathsf{VER}(m,z)$ to obtain

$$\log_\sigma m - \log_\sigma \log_\sigma \frac{m}{\ell+1} \leq m/(\ell+1).$$

This gives a maximum value for ℓ, that is

$$\log_\sigma \frac{m}{\log_\sigma \frac{m}{\ell+1}} \leq m/(\ell+1), \text{ which gives } \ell < m\left(\frac{1}{\log_\sigma \frac{m}{\log_\sigma \frac{m}{\ell+1}}}\right).$$

Therefore we get the following second condition on ℓ, simplified slightly for comprehension,

$$\ell < m\left(\frac{\log\sigma}{\log m + \log\log\sigma - \log(\log m - \log(\ell+1))}\right).$$

This gives a third condition on ℓ, tighter with respect to $\ell < m$, namely, $\ell < \frac{m-1}{2}$. By Lemma 2.2.4, we have that $\ell \leq z\log z$, and so from the above we obtain

$$\frac{z}{m} < \min\left\{\frac{1}{2\log z + \frac{1}{z}}, \frac{\log\sigma}{\log z(\log m + \log\log\sigma)}\right\}.$$

□

Example 2.2.2. Running example for algorithm WTM

Given a cumulative weight threshold $1/z = 0.25$ where $z = 4$, a string $x = \texttt{actactaatc}$ of length $m = 10$ as the pattern and a weighted string Y of length $n = 20$ on alphabet $\Sigma = \{\texttt{a},\texttt{c},\texttt{g},\texttt{t}\}$:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$\pi_i(\texttt{a})$	1	0.5	0	1	0	0	1	0.5	0	0	1	0	0	1	0	0	0.5	1	0	0
$\pi_i(\texttt{c})$	0	0.5	0	0	1	0	0	0.5	0	1	0	1	0	0	0.9	0	0	0	0	1
$\pi_i(\texttt{g})$	0	0	0	0	0	0	0	0	0	0	0	0	0.5	0	0.1	0	0.5	0	0	0
$\pi_i(\texttt{t})$	0	0	1	0	0	1	0	0	1	0	0	0	0.5	0	0	1	0	0	1	0

From Y we can construct

$$\begin{array}{cccccccccccccccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \\ y' = & \texttt{a} & \lambda & \texttt{t} & \texttt{a} & \texttt{c} & \texttt{t} & \texttt{a} & \lambda & \texttt{t} & \texttt{c} & \texttt{a} & \texttt{c} & \lambda & \texttt{a} & \texttt{c} & \texttt{t} & \lambda & \texttt{a} & \texttt{t} & \texttt{c} \end{array}$$

$\ell = 4$ so we partition x into 5 fragments: $f_0 = \texttt{ac}$, $f_1 = \texttt{ta}$, $f_2 = \texttt{ct}$, $f_3 = \texttt{aa}$ and $f_4 = \texttt{tc}$. After searching all fragments in y', we have: f_0 occurs at position 10, 13; f_1 occurs at position 2; f_2 occurs at position 4, 14; f_3 does not occur in y'; f_4 occurs at position 8, 18. Only the occurrence of f_0 at position 13 cannot be extended, and after extension we have two factor of Y need to verify: $Y[0..9]$ and $Y[10..19]$:

$$\prod_{i=0}^{9} \pi_i(x[i]) = 0.5 \times 0.5 = 0.25 = 1/z$$

$$\prod_{i=0}^{9} \pi_{i+10}(x[i]) = 0.5 \times 0.5 \times 0.9 = 0.225 < 1/z$$

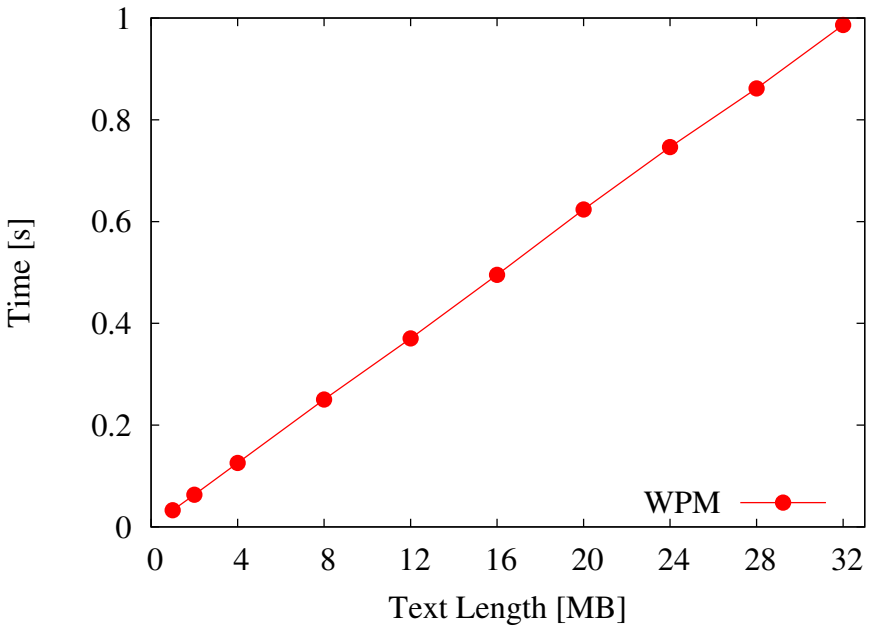
As a result we report position 0 to be the staring position of an occurrence of x in Y.

2.2.4 Experimental Results

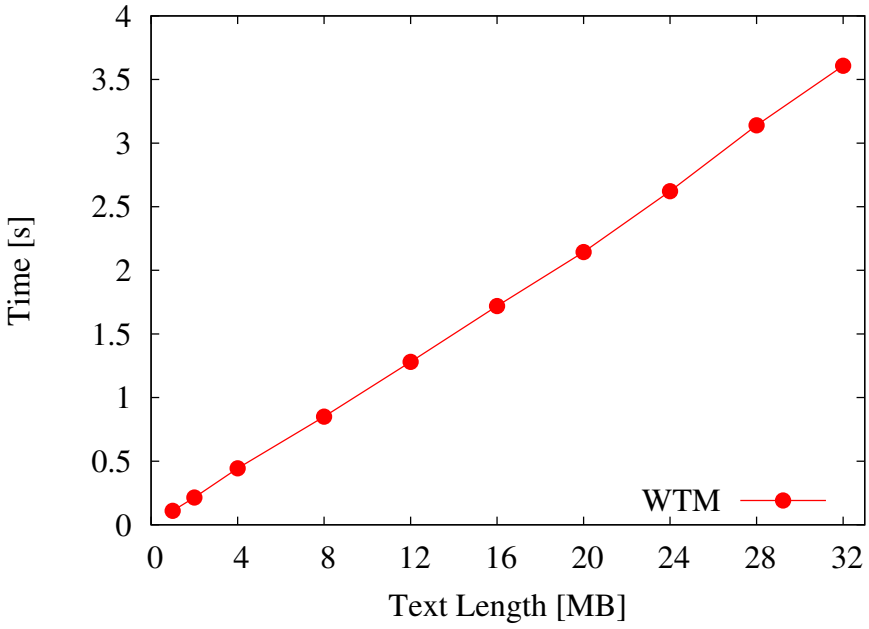
Algorithms WPM and WTM were implemented as a program to perform weighted string matching. The program was implemented in the C++ programming language and developed under the GNU/Linux operating system. The input parameters for WPM are a weighted pattern in the form of a probability matrix, a text, and a cumulative weight threshold. The input parameters for WTM are a pattern, a weighted text, and a cumulative weight threshold. The output is the positions of all the occurrences. The implementation is distributed under the GNU General Public License (GPL), and it is available at http://github.com/YagaoLiu/FWSM. Our experiments were conducted on a PC using one core of Intel Core i5-4690 CPU at 3.50GH under GNU/Linux. All programs referred to in this section were compiled with g++ version 4.8.4 at optimisation level 3 (-O3). As the most commonly used data of this type are molecular sequences, we used DNA data in our experiments. (The larger the alphabet, the better our algorithms perform, and in this case $\sigma = 4$.)

In this experiment our task was to establish that the elapsed time of our implementations grows *in practice* linearly with respect to the size of the text. Synthetic datasets were generated using a randomised script (uniform distribution). One pattern and ten texts with different lengths were used for each algorithm. For WPM, we used one weighted pattern of length 128 with a black positions percentage $\delta = 4\%$, and ten texts of lengths ranged from 1MB to 32MB. For WTM, we used a pattern of length 128 and ten weighted texts of length 1MB to 32MB with a uncertain positions percentage $\delta = 4\%$. $1/z = 1/16$ was used in all runs. These lengths of patterns and texts are set to keep a balance between the time and the computer memory used, and similar parameters are used in the experiments in the rest of this

thesis. The results are plotted in Fig. 2.1. It becomes evident from the results that the elapsed time of our implementations grows linearly with respect to the size of the text.



(a) Elapsed time of WPM



(b) Elapsed time of WTM

Fig. 2.1 Elapse time of WPM and WTM on synthetic DNA datasets with $1/z = 1/16$ and $\delta = 4\%$.

2.3 On-line $\mathcal{O}(\frac{nz\log m}{m})$-time Pattern Matching Algorithm and $\mathcal{O}(\frac{nz(\log m+k)}{m})$-time Approximate Pattern Matching Algorithm on Weighted Sequences

2.3.1 Our Contribution

In this section, We present three new on-line algorithms: one to solve problem WEIGHTEDTEXTMATCHING, one to solve problem GENERALWEIGHTEDPATTERNMATCHING and the last one to solve problem APPROXWEIGHTEDTEXTMATCHING.

We also provide their *average-case* analysis, obtaining the following results. Note that preprocessing resources below denote worst-case complexities, $0 < c < 1/2$ is an absolute constant, $v = \frac{2^\sigma - 1}{2^{\sigma-1}}$, $d = 1 + (1-c)\log_v(1-c) + c\log_v c$, and $a = 4\sqrt{c(1-c)}$.

Problem	Preprocessing space	Preprocessing time	Search time	Conditions
WTM	$\mathcal{O}(m)$	$\mathcal{O}(m)$	$\mathcal{O}(\frac{nz\log m}{m})$	
GWPM	$\mathcal{O}(zm)$	$\mathcal{O}(zm)$	$\mathcal{O}(\frac{nz\log m}{m})$	
AWTM	$\mathcal{O}(\sigma^q)$	$\mathcal{O}(mq\sigma^q)$	$\mathcal{O}(\frac{nz(\log m+k)}{m})$	$q \geq \frac{3\log_v m - \log_v a}{d}$, $\frac{k}{m} \leq c - \frac{2cq}{m}$

Our computational model. We assume word-RAM model with word size $w = \Omega(\log(nz))$. We consider the log-probability model of representations of weighted strings in which probability operations can be realised exactly in $\mathcal{O}(1)$ time. We assume that $\sigma = \mathcal{O}(1)$ since the most commonly studied alphabet is $\{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}$. In this case a weighted string of length n

has a representation of size $\mathcal{O}(n)$. A position on a weighted string is viewed as a non-empty subset of the alphabet such that each letter of this subset has probability of occurrence greater than 0. For the analysis, we assume all possible non-empty subsets of the alphabet are independent and identically distributed random variables uniformly distributed.

2.3.2 Tools for Standard and Weighted Strings

Suffix trees are used as computational tools. The *suffix tree* of a non-empty standard string y is denoted by $\mathcal{T}(y)$.

We next define some primitive operations on weighted strings. Suppose we are given a cumulative weight threshold $1/z$. Let u be a string of length m and v be a weighted string of length m. We define operation $\mathsf{VER1}(u,v,z)$: it returns *true* if $u =_z v$ and *false* otherwise. Let u be a weighted string of length m and v be a weighted string of length m. We define operation $\mathsf{VER2}(u,v,z)$: it returns *true* if $u =_z v$ and *false* otherwise. For a weighted string v, by $\mathcal{H}(v)$ we denote a string obtained from v by choosing at each position the heaviest letter, that is, the letter with the maximum probability (breaking ties arbitrarily). We call $\mathcal{H}(v)$ the *heavy string* of v. Let u be a string of length m and v be a weighted string of length m. Given a non-negative integer $k < m$, we can check whether $u =_{z,k} v$ using Function VER3. An implementation of VER3 is provided below. Intuitively, we replace at most k letters of u with the heaviest letter in v based on how much these letter replacements contribute to the cumulative probability.

Function *VER3*(u,v,z,k)

$v' \leftarrow \mathcal{H}(v)$;

$A \leftarrow$ EMPTYLIST();

foreach *i such that* $u[i] \neq v'[i]$ **do**

if $\pi_i(u[i]) = 0$ **then**

$u[i] \leftarrow v'[i]$;

$k \leftarrow k-1$;

if $k < 0$ **then**

return *false*;

else

$\alpha \leftarrow \pi_i(v'[i])/\pi_i(u[i])$;

$A \leftarrow$ INSERT$(< i, \alpha >)$;

Find the kth largest element in A with respect to α;

Add the k largest elements of A with respect to α to set A_k;

foreach $< i, \alpha > \in A_k$ **do**

$u[i] \leftarrow v'[i]$;

if $\prod_{j=0}^{m-1} \pi_j(u[j]) \geq 1/z$ **then**

return *true*;

return *false*;

Example 2.3.1. Running example for VER3

Given a cumulative weight threshold $1/z = 0.2$, an integer $k = 2$, a weighted string V of length 7:

i	0	1	2	3	4	5	6
$\pi_i(\mathtt{a})$	0.6	1	0	0.1	0.2	0	0
$\pi_i(\mathtt{c})$	0	0	1	0	0.2	0	0
$\pi_i(\mathtt{g})$	0.4	0	0	0.9	0	1	0
$\pi_i(\mathtt{t})$	0	0	0	0	0.6	0	1

and given a string $u = \mathtt{gacgcga}$, we try to verify if $u =_{z,k} V$.

First We construct the heavy string of V: $v' = \mathtt{aacgtgt}$. Comparing v' against u, there exist three mismatch positions: $0,4,6$. Since $\pi_6(a) = 0$, we replace $u[6]$ by $v'[6]$, then $u = \mathtt{gacgcgt}$ and $k = 1$. For position 0 and 4, we insert pair $< 0, 1.5 >$ and $< 4, 3 >$ to A, and then add $< 4, 3 >$ to A_k. We replace $u[4]$ by $v'[4]$ and have $u = \mathtt{gacgtgt}$. Finally we compute the cumulative occurrence probability:

$$\prod_{i=0}^{7} \pi_i(u[i]) = 0.4 \times 0.9 \times 0.6 = 0.216 > 1/z$$

We return that $u =_{z,k} V$ is true.

Lemma 2.3.1. *VER1, VER2, and VER3 can be implemented to work in time $\mathcal{O}(m)$, $\mathcal{O}(mz)$, and $\mathcal{O}(m)$, respectively.*

Proof. For VER1 we can check whether $u =_z v$ in time $\mathcal{O}(m)$ by checking $\prod_{j=0}^{m-1} \pi_j(u[j]) \geq 1/z$. For VER2 we can check whether $u =_z v$ in time $\mathcal{O}(mz)$ using the algorithm of [12].

Let us denote by $E = \{e_1, \ldots, e_{|E|}\}$, $|E| \leq k$, the set of positions of the input string u, which we replace in VER3 by the heaviest letter of v. We denote the resulting string by u'. Towards contradiction, assume we can guess a set $F = \{f_1, \ldots, f_{|F|}\}$, $|F| \leq k$, of positions over u resulting in another string u'' such that $P_{u''} = \prod_{j=0}^{m-1} \pi_j(u''[j]) > P_{u'} = \prod_{j=0}^{m-1} \pi_j(u'[j]) \geq P_u = \prod_{j=0}^{m-1} \pi_j(u[j])$. It must hold that

$$\frac{P_{u''}}{P_u} = \frac{\pi_{f_1}(u''[f_1]) \ldots \pi_{f_{|F|}}(u''[f_{|F|}])}{\pi_{f_1}(u[f_1]) \ldots \pi_{f_{|F|}}(u[f_{|F|}])} > \frac{P_{u'}}{P_u} = \frac{\pi_{e_1}(u'[e_1]) \ldots \pi_{e_{|E|}}(u'[e_{|E|}])}{\pi_{e_1}(u[e_1]) \ldots \pi_{e_{|E|}}(u[e_{|E|}])}.$$

In case $E = F$, this implies that there exist letters heavier than the corresponding heaviest letters, a contradiction. In case $E \neq F$, given that there exists no letter heavier than the heaviest letter at each position, this implies that there exists an element $< i, \alpha >$, $i \in F$, in list A which is the rth largest, $r \leq k$, with respect to α, and it is larger than the rth element picked by VER3, a contradiction. All operations in VER3 can be trivially done in time $\mathcal{O}(m)$ except for finding the kth largest element in a list of size $\mathcal{O}(m)$, which can be done in time $\mathcal{O}(m)$ using the *introselect* algorithm [52]. □

We say that u is a *(right-)maximal factor* of a weighted string x at position i if u is a valid factor of x starting at position i and no string $u' = u\alpha$, for $\alpha \in \Sigma$, is a valid factor of x at this position.

Fact 2.3.2 ([7]). *A weighted string has at most z different maximal factors starting at a given position.*

2.3.3 Algorithms

In this section, we view a weighted string as an indeterminate string in order to conduct the average-case analysis of the algorithms according to our model. Additionally, we refer to any indeterminate string of length q as *indeterminate q-gram*.

Weighted Text Matching

In this section we present a remarkably simple and efficient algorithm to solve problem WEIGHTEDTEXTMATCHING. We start by providing a lemma on the probability that a random indeterminate q-gram and a standard q-gram match.

Lemma 2.3.3. *Let u be a standard q-gram and v be a uniformly random indeterminate q-gram. The probability that $u \approx v$ is no more than $(\frac{2^{\sigma-1}}{2^{\sigma}-1})^q$, which tends to $\frac{1}{2^q}$ as σ increases.*

Proof. There are less than 2^{σ} non-empty subsets of an alphabet of size σ. Let $a \in \Sigma$, then $2^{\sigma-1}$ of these subsets include a. Clearly then the probability that two positions of u and v match is no more than $\frac{2^{\sigma-1}}{2^{\sigma}-1}$. Therefore the probability of them matching at every position is no more than $(\frac{2^{\sigma-1}}{2^{\sigma}-1})^q$. □

Algorithm $qWTM(x,m,y,n,z,q)$

Construct $\mathcal{T}(x)$;

$i \leftarrow 0$;

while $i < n-m+1$ **do**

$j \leftarrow i+m-q$;

Let $\mathcal{A}$ denote the set of all valid q-grams starting at position j in y;

foreach $s \in \mathcal{A}$ **do**

Check if s occurs in x using $\mathcal{T}(x)$;

if *no* $s \in \mathcal{A}$ *occurs in* x **or** $\mathcal{A} = \emptyset$ **then**

$i \leftarrow j+1$;

else

if $VER1(x, y[i..i+m-1], z) = true$ **then**

output i;

$i \leftarrow i+1$;

Theorem 2.3.4. *Algorithm qWTM solves problem* WEIGHTEDTEXTMATCHING *in average-case search time* $\mathcal{O}(\frac{nz\log m}{m})$ *if we set* $q \geq 3\log_{\frac{2^\sigma-1}{2^\sigma-1}} m$, *which tends to* $3\log_2 m$ *as* σ *increases. The worst-case preprocessing time and space is* $\mathcal{O}(m)$.

Proof. By Fact 1.3.1 the time and space required for constructing $\mathcal{T}(x)$ is $\mathcal{O}(m)$. We consider a sliding window of size m of y and read q-grams backwards from the end of this window and check if they occur anywhere within x. By Fact 1.3.1 this check can be done in time $\mathcal{O}(q)$ per

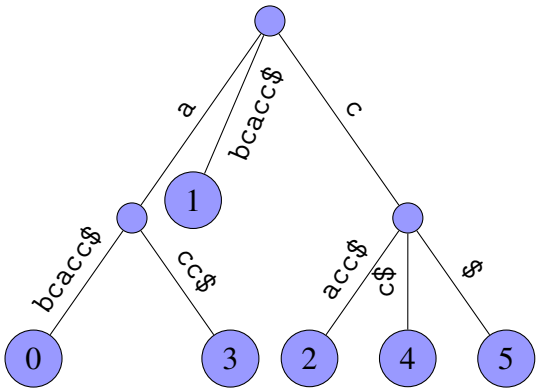
q-gram. If a q-gram occurs anywhere in x then we verify the entire window, otherwise we shift the window $m-q+1$ positions to the right. Clearly none of the skipped positions can be the starting position of any occurrence of x as if this was the case, the q-gram must occur in x; so the algorithm is correct. Verifying (all starting positions of) the window takes time $\mathcal{O}(m^2)$ by Lemma 2.3.1 and the probability that a q-gram matches within a pattern of length $m > q$ is no more than $m(\frac{2^{\sigma-1}}{2^{\sigma}-1})^q$ by Lemma 2.3.3. We note that reading the q-grams takes time $\mathcal{O}(zq)$ per position by Fact 2.3.2, so to achieve the claimed runtime we must pick a value for q such that the expected cost per window is $\mathcal{O}(zq)$. This is achieved when $\mathcal{O}(m^3(\frac{2^{\sigma-1}}{2^{\sigma}-1})^q) = \mathcal{O}(zq)$. It is always true when $q \geq 3\log_{\frac{2^{\sigma-1}}{2^{\sigma}-1}} m$, which tends to $3\log_2 m$ as σ increases. There are $\mathcal{O}(\frac{n}{m})$ non-overlapping windows of length m and this proves the theorem. □

Example 2.3.2. Running example for algorithm qWTM

Assuming we are given threshold $1/z = 1/4$. Given a weighted string Y of length $n = 15$ as a text :

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$\pi_i(\mathtt{a})$	1	0.5	1	0	0.5	1	1	0.5	0	0.5	0	0	0	1	0
$\pi_i(\mathtt{b})$	0	0	0	1	0.5	0	0	0.5	0	0	0	0	1	0	0
$\pi_i(\mathtt{c})$	0	0.5	0	0	0	0	0	0	1	0.5	1	1	0	0	1

and given a string $x = \mathtt{abcacc}$ of length $m = 6$ as pattern. For this example, we set $q = 2$. First we construct suffix tree of x:



We begin from $i = 0$, $j = 4$ and $Y[4..5] =$ `(a,0.5),(b,0.5)]a`. There exist two q-grams: `aa` and `ba` at this position. We search both q-grams in x and no occurrence exists. At this moment we shift i to position 5 and $j = 9$, $Y[9..10] =$ `[(a,0.5),(c,0.5)c]`. There exist two q-grams `ac` and `cc`. We find both q-grams in x but $\mathsf{VER1}(x, Y[5..10])$ returns *false*. We shift i by 1 and $j = 10$, $Y[10..11] =$ `cc`. We search and find the only q-gram `cc` in x, and $\mathsf{VER1}(x, Y[6..11])$ turns *true*. Thus we report position 6 as an occurrence. Then we shift to $i = 7$, $j = 11$, $Y[11..12] =$ `cb` and q-gram is `cb`. This q-gram do not occur in x, so we shift i to $i = 12$. Since $i > n - m + 1 = 10$, we reach the end of this algorithm.

General Weighted Pattern Matching

In this section we present an algorithm, denoted by GWPM, to solve problem GENERAL-WEIGHTEDPATTERNMATCHING. Algorithm GWPM largely follows algorithm qWTM.

Lemma 2.3.5. *Let U and V be uniformly random indeterminate q-grams. The probability that $U \approx V$ is no more than $(1-(1-(\frac{2^{\sigma-1}}{2^{\sigma}-1})^2)^{\sigma})^q$, which tends to $(1-(\frac{3}{4})^{\sigma})^q$ as σ increases.*

Proof. There are less than 2^{σ} non-empty subsets of an alphabet of size σ. Let $a \in \Sigma$, then $2^{\sigma-1}$ of these subsets include a. Clearly then the probability that a occurs at both positions is

no more than $(\frac{2^{\sigma-1}}{2^{\sigma}-1})^2$. It then follows that the probability that the two sets have a non-empty intersection is no more than $1-(1-(\frac{2^{\sigma-1}}{2^{\sigma}-1})^2)^{\sigma}$. Therefore the probability that all positions have a non-empty intersection is no more than $(1-(1-(\frac{2^{\sigma-1}}{2^{\sigma}-1})^2)^{\sigma})^q$. □ □

Theorem 2.3.6. *Algorithm GWPM solves problem* GENERALWEIGHTEDPATTERNMATCHING *in average-case search time* $\mathcal{O}\left(\frac{nz\log m}{m}\right)$ *if we set* $q \geq 3\log_u m$, *where* $u=\frac{(2^{\sigma}-1)^{2\sigma}}{(2^{\sigma}-1)^{2\sigma}-((2^{\sigma}-1)^2-2^{2\sigma-2})^{\sigma}}$, *which tends to* $3\log_{1/(1-(\frac{3}{4})^{\sigma})} m$ *as* σ *increases. The worst-case preprocessing time and space is* $\mathcal{O}(zm)$.

Proof. Let $x_1, x_2, \ldots, x_f$ denote all f valid factors of length m of X. We construct string $X' = x_1\$x_2\$\ldots\$x_f$, where $\$ \notin \Sigma$. By Facts 1.3.1 and 2.3.2 the time and space required for constructing $\mathcal{T}(X)$ is $\mathcal{O}(zm)$. Plugging Lemmas 2.3.1 and 2.3.5 to the proof of Theorem 2.3.4 yields the result. This is achieved when $\mathcal{O}(m^3 z(1-(1-(\frac{2^{\sigma-1}}{2^{\sigma}-1})^2)^{\sigma})^q) = \mathcal{O}(zq)$. It is always true when $q \geq 3\log_u m$, where $u=\frac{(2^{\sigma}-1)^{2\sigma}}{(2^{\sigma}-1)^{2\sigma}-((2^{\sigma}-1)^2-2^{2\sigma-2})^{\sigma}}$, which tends to $3\log_{1/(1-(\frac{3}{4})^{\sigma})} m$ as σ increases. □

Algorithm *GWPM*$(X, m, Y, n, z, q, \Sigma)$

Let $\mathcal{F} = \{x_1, \ldots, x_f\}$ denote the set of all valid factors of length m of X;

if $\mathcal{F} = \emptyset$ **then**

return;

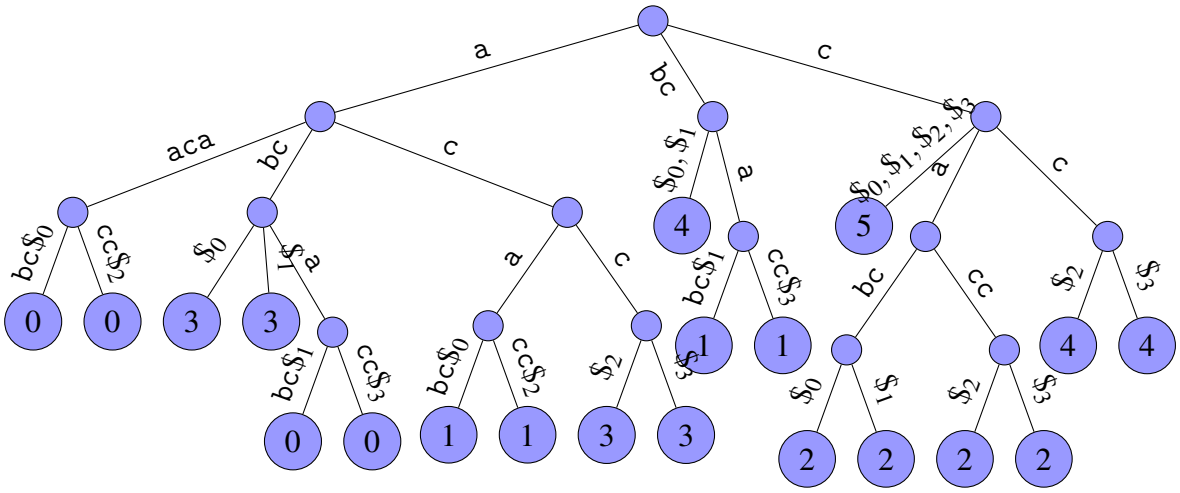
Construct string $x' = x_1\$\ldots\$x_f$, where $\$ \notin \Sigma$;

The rest of this algorithm follows the steps in qWTM, where we use x' instead of x and VER2 for verification.

Example 2.3.3. Given a threshold $1/z = 1/4$ and a weighted string of length 6 as pattern:

$$X = \texttt{a[(a,0.5),(b,0.5)]ca[(b,0.5),(c,0.5)]c}$$

we construct $x' = \texttt{aacabc}\$_0\texttt{abcabc}\$_1\texttt{aacacc}\$_2\texttt{abcacc}\$_3$, and $\mathcal{T}(x')$:



The rest of this algorithm can just follow the steps in running example 2.3.2.

Approximate Weighted Text Matching

In this section we present an algorithm to solve problem ApproxWeightedTextMatching. The algorithm, denoted by AWTM, is split into two distinct stages: preprocessing the pattern x and searching the weighted text Y.

Preprocessing. We build a q-gram index in a similar way as that proposed by Chang and Marr in [19]. Intuitively, we wish to determine the minimum possible Hamming distance (mismatches) between every q-gram on Σ and any q-gram of x. An index like this allows us

to lower bound the Hamming distance between a window of y and x without computing the Hamming distance between them. To build this index, we generate every string of length q on Σ, and find the minimum Hamming distance between it and all factors of length q of x. This information can easily be stored by generating a numerical representation of the q-gram and storing the minimum Hamming distance in an array at this location. If we know the numerical representation, we can then look up any entry in constant time. This index has size $\mathcal{O}(\sigma^q)$ and can be trivially constructed in worst-case time $\mathcal{O}(mq\sigma^q)$ and space $\mathcal{O}(\sigma^q)$.

Lemma 2.3.7. *Let u be a standard q-gram and v be a uniformly random indeterminate q-gram. The probability that u and v match with cq-mismatches is at most*

$$\left(\frac{2^{\sigma-1}}{2^\sigma-1}\right)^q (1-c)^{-(1-c)q} c^{-cq} \frac{1}{4\sqrt{c(c-1)}}$$

, where $0 < c < 1/2$.

Proof. Without loss of generality assume that cq is an integer. By Lemma 2.3.3, the probability the q-grams match with *exactly* i mismatches is

$$\binom{q}{i}\left(\frac{2^{\sigma-1}}{2^\sigma-1}\right)^q.$$

Therefore, by Lemma 3.8.2 in [47] on the sum of binomial coefficients, the probability that u and v match with cq-mismatches is

$$\sum_{i=0}^{cq}\binom{q}{i}\left(\frac{2^{\sigma-1}}{2^{\sigma}-1}\right)^{q}=\left(\frac{2^{\sigma-1}}{2^{\sigma}-1}\right)^{q}\sum_{i=0}^{cq}\binom{q}{i}\leq\left(\frac{2^{\sigma-1}}{2^{\sigma}-1}\right)^{q}(1-c)^{-(1-c)q}c^{-cq}\frac{1}{4\sqrt{c(c-1)}}.$$

This decreases exponentially in q when $(1-c)^{-(1-c)}c^{-c}>0$ which holds for $0<c<1/2$. It tends to $2^{-q}(1-c)^{-(1-c)q}c^{-cq}\frac{1}{4\sqrt{c(c-1)}}$ as σ increases. □ □

Searching. We wish to read backwards enough indeterminate q-grams from a window of size m such that the probability that we must verify the window is small and the amount that we can shift the window by is sufficiently large. By Lemma 2.3.7, we know that the probability of a random indeterminate q-gram occurring in a string of length m with cq-mismatches is no more than $m\left(\frac{2^{\sigma-1}}{2^{\sigma}-1}\right)^{q}(1-c)^{-(1-c)q}c^{-cq}\frac{1}{4\sqrt{c(1-c)}}$. For the rest of the discussion let $a=4\sqrt{c(1-c)}$.

In the case when we read $k/(cq)$ indeterminate q-grams, we know that with probability at most $(k/(cq))m\left(\frac{2^{\sigma-1}}{2^{\sigma}-1}\right)^{q}(1-c)^{-(1-c)q}c^{-cq}1/a$ we have found at most k mismatches. This does not permit us to discard the window if all q-grams occur with at most cq mismatches. To fix this, we instead read $\ell=1+k/(cq)$ q-grams. If any indeterminate q-gram occurs with less than cq mismatches, we will need to verify the window; but if they all occur with at least cq mismatches, we must exceed the threshold k and can shift the window. When shifting the window we have the case that we shift after verifying the window and the case that the mismatches exceed k so we do not verify the window. If we have verified the window, we can shift past the last position we checked for an occurrence: we can shift by m positions. If we have not verified the window, as we read a fixed number of indeterminate

q-grams, we know the minimum-length shift we can make is one position past this point. The length of this shift is at least $m-(q+k/c)$ positions. This means we will have at most $\frac{n}{m-(q+k/c)} = \mathcal{O}(\frac{n}{m})$ windows. The previous statement is only true assuming $m > q+k/c$, as then the denominator is positive. From there we see that we also have the condition that $q+k/c$ can be at most εm, where $\varepsilon < 1$, so the denominator will be $\mathcal{O}(m)$. This puts the condition on c, that is, $c > \frac{k}{\varepsilon m-q}$. Therefore, for each window, we verify with probability at most $(1+k/(cq))m\left(\frac{2^{\sigma-1}}{2^{\sigma}-1}\right)^{q}(1-c)^{-(1-c)q}c^{-cq}1/a$. So the probability that a verification is triggered is

$$(1+k/(cq))m\left(\frac{2^{\sigma-1}}{2^{\sigma}-1}\right)^{q}(1-c)^{-(1-c)q}c^{-cq}1/a.$$

By Lemma 2.3.1 verification takes time $\mathcal{O}(m^2)$; per window the expected cost is

$$\mathcal{O}(m^2)(1+k/(cq))m\left(\frac{2^{\sigma-1}}{2^{\sigma}-1}\right)^{q}(1-c)^{-(1-c)q}c^{-cq}1/a =$$

$$\mathcal{O}\left(\frac{(q+k)m^3\left(\frac{2^{\sigma-1}}{2^{\sigma}-1}\right)^{q}(1-c)^{-(1-c)q}c^{-cq}}{qa}\right).$$

We wish to ensure that the probability of verifying a window is small enough that the average work done is no more than the work we must do if we skip a window without verification. When we do not verify a window, we read $\ell = 1+k/(cq)$ indeterminate q-grams and shift the window. This means that we process $q+k/c = \mathcal{O}(q+k)$ positions. So a sufficient *condition* is the following:

$$\frac{(q+k)m^3\left(\frac{2^{\sigma-1}}{2^{\sigma}-1}\right)^{q}(1-c)^{-(1-c)q}c^{-cq}}{qa} = \mathcal{O}(q+k).$$

For sufficiently large m, this holds if we set $q \geq \frac{3\log_v m-\log_v a-\log_v q}{1+(1-c)\log_v(1-c)+c\log_v c}$, where $v = \frac{2^\sigma-1}{2^{\sigma-1}}$, which tends to $\frac{3\log_2 m-\log_2 a-\log_2 q}{1+(1-c)\log_2(1-c)+c\log_2 c}$ as σ increases.

Algorithm *AWTM*$(x,m,Y,n,z,k,q,\ell,\Sigma)$

$\mathsf{D}[0\,..\,|\Sigma|^q-1]\leftarrow 0$;

foreach $s\in\Sigma^q$ **do**

Compute the *minimal* Hamming distance e between s and any factor of x,

and set $\mathsf{D}[s]\leftarrow e$;

$i\leftarrow 0$;

while $i<n-m+1$ **do**

$d\leftarrow 0$;

foreach $t\in[1,\ell]$ **do**

$j\leftarrow i+m-t\times q$;

Let $\mathcal{A}$ denote the set of all valid q-grams starting at position j in Y;

if $\mathcal{A}=\emptyset$ **then break** ;

else

$d_{\min}\leftarrow q$;

foreach $s\in\mathcal{A}$ **do**

$d_{\min}\leftarrow\min\{d_{\min},\mathsf{D}[s]\}$;

$d\leftarrow d+d_{\min}$;

if $d>k$ **or** $\mathcal{A}=\emptyset$ **then**

$i\leftarrow j+1$;

else

if *VER3*$(x,Y[i\,..\,i+m-1],z,k)=$ *true* **then**

output i;

$i\leftarrow i+1$;

For this analysis to hold we must be able to read the required number of indeterminate q-grams. Note that the above probability is the probability that at least one of q-grams matches with fewer than cq mismatches. To ensure we have enough unread random q-grams for the above analysis to hold, the window must be of size $m \geq 2q + 2k/c$. Consider the case where $2q + 2k/c > m \geq 2q + k/c$. If we have verified a window then we have enough new random q-grams, and if we have just shifted, then we know that all the q-grams we previously read matched with at least cq mismatches and we have at least one new q-gram. The probability that one of these matches with less than cq mismatches is less than the one used above so the analysis holds in both cases. Note that this technique can work for any ratio which satisfies $k/m \leq c - \frac{2cq}{m}$.

Now recall that in fact we are processing a *weighted* string as a text, not an indeterminate one. By the aforementioned analysis, we can choose a suitable value for c and q, to obtain the following result, noting also that it takes time $\mathcal{O}(z(q+k))$, by Fact 2.3.2, to obtain the *valid* q-grams of $\mathcal{O}(q+k)$ positions of the weighted text.

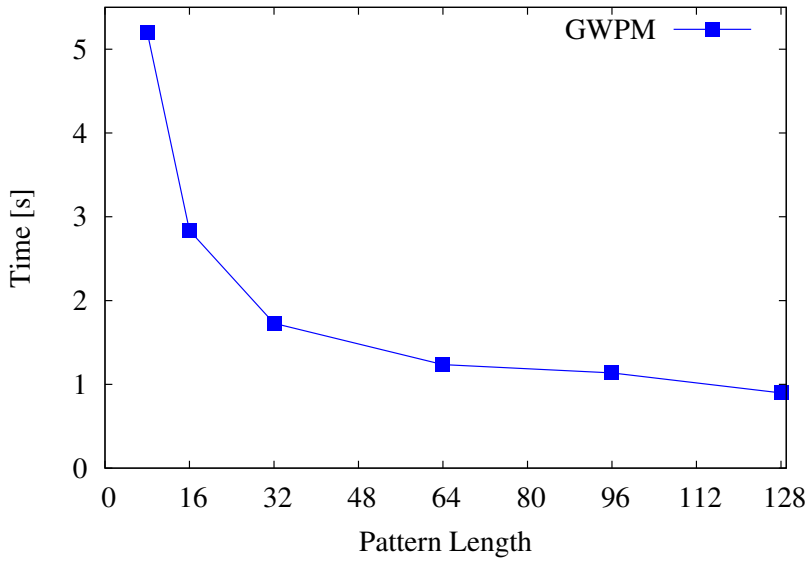
Theorem 2.3.8. *Algorithm AWTM solves problem* ApproxWeightedTextMatching *in average-case search time* $\mathcal{O}\left(\frac{nz(\log m+k)}{m}\right)$ *if* $k/m \leq c - \frac{2cq}{m}$ *and we set* $q \geq \frac{3\log_v m - \log_v a}{1+(1-c)\log_v(1-c)+c\log_v c}$, *where* $v = \frac{2^\sigma-1}{2^\sigma-1}$, $a = 4\sqrt{c(1-c)}$ *and* $0 < c < 1/2$, *which tends to* $\frac{3\log_2 m - \log_2 a}{1+(1-c)\log_2(1-c)+c\log_2 c}$ *as* σ *increases. The worst-case preprocessing time and space is* $\mathcal{O}(mq\sigma^q)$ *and* $\mathcal{O}(\sigma^q)$, *respectively.*

2.3.4 Experimental Results

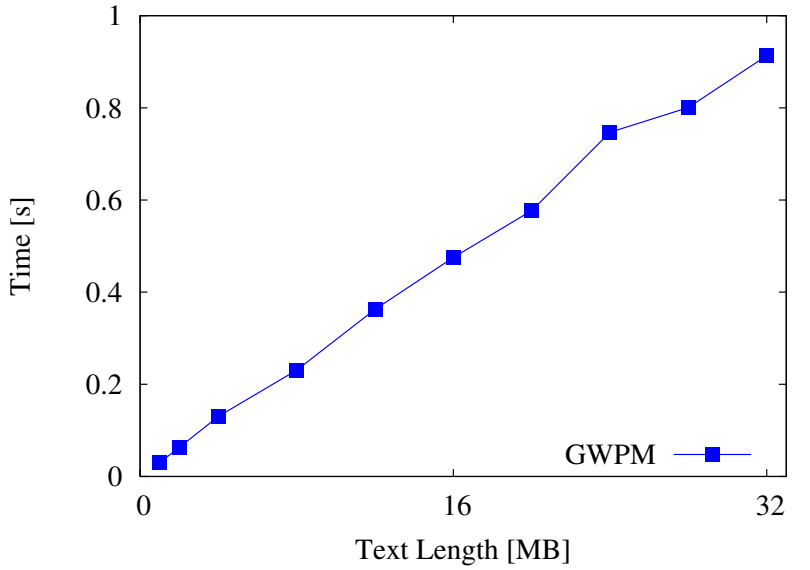
Algorithms qWTM and GWPM were implemented as one program to perform exact weighted string matching. Algorithm AWTM was implemented as a program to perform approximate weighted string matching. The programs were implemented in the C++ programming language and developed under the GNU/Linux operating system. The input parameters for qWTM and GWPM are a pattern (weighted string for GWPM), a weighted text, and a cumulative weight threshold. The output of these programs is the starting positions of all valid occurrences. The input parameters for AWTM are a pattern, a weighted text, a cumulative weight threshold, and an integer k for mismatches. The output of this program is the starting positions of all valid occurrences within k mismatches. These implementations are distributed under the GNU General Public License (GPL). the implementation for qWTM and GWPM is available at https://github.com/YagaoLiu/GWSM, and the implementation for AWTM is available at https://github.com/YagaoLiu/HDwtm. The experiments were conducted on a Desktop PC using one core of Intel Core i5-4690 CPU at 3.50GHz under GNU/Linux. All programs were compiled with g++ version 4.8.4 at optimisation level 3 (-O3). As the most commonly used data of this type are molecular sequences we used DNA data in our experiments. (The larger the alphabet, the better WPM and WTM perform, and in this case $\sigma = 4$.)

In the first experiment, our task was to evaluate the time performance of GWPM. Synthetic datasets were used in the experiments. For GWPM, we first used 5 weighted patterns of length 8 to 128 and weighted text of length 32MB with uncertain positions percentage $\delta = 4\%$. We then used one weighted pattern of length 128 and ten weighted text of lengths

1MB to 32MB with $\delta = 4\%$. The cumulative weight threshold was set to $z = 16$, and q was simply set to 4. The results are plotted in Fig 2.2.



(a) Elapsed time of GWPM using patterns of length 1 to 128 and weighted text of length 32MB.

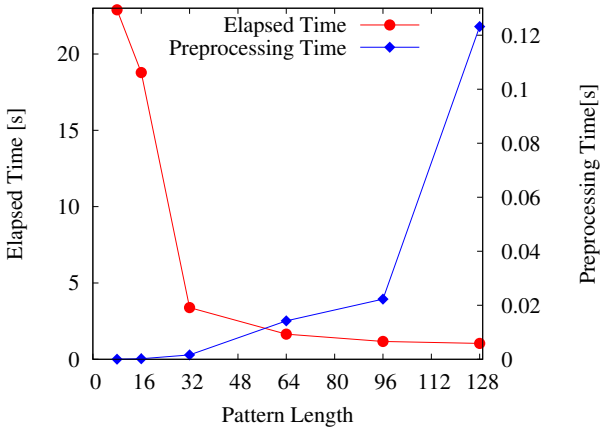


(b) Elapsed time of GWPM using pattern of length 128 and weighted text of length 1MB to 32MB.

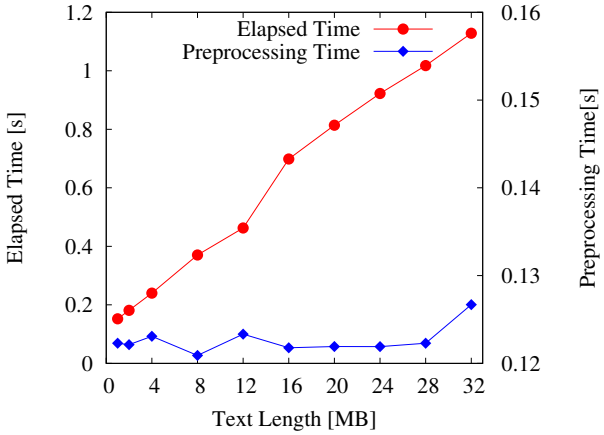
Fig. 2.2 Elapsed time of GENERALWEIGHTEDPATTERNMATCHING with $z = 16$ and $\delta = 4\%$ on synthetic DNA datasets.

From the figures, we observe that: (i)as the pattern length increases, the elapsed time of GWPM decreases; (ii)the elapsed time grows linearly with respect to the length of texts. This results proves our theorem: GWPM solves problem GENERALWEIGHTEDPATTERNMATCHING in average-case search time $\mathcal{O}\Big(\frac{nz\log m}{m}\Big)$.

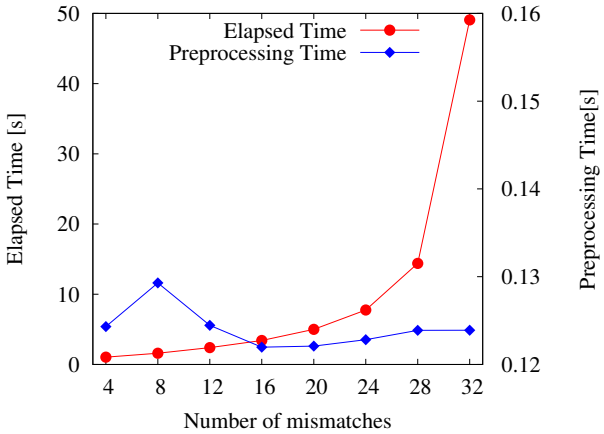
In the second experiments, our task was to evaluate the time performance of AWTM. Similar with the experiments of GWPM, we first used 5 patterns of length 8 to 128 and a weighted text of length 32MB with $\delta = 4\%$, and we set $k = 4$ for the mismatches. Then we used one pattern of length 128 and weighted text of length 1MB to 32MB with $\delta = 4\%$ and $k = 4$. We also applied an experiment using a pattern of length 128 and a weighted text of length 32MB, and with k ranging from 4 to 32. The cumulative weight threshold was set to $z = 16$ and for a pattern of length m, q was set to $q = log(m)$. Both the elapsed time and the preprocessing time of our algorithm are plotted in Fig 2.3. From the figures we observe that: (i)the time decreases when the length of patterns increases; (ii)the time increases linearly with the length of the texts; (iii)the time increases when given a larger k; (iv)the preprocessing time is only effected by the value of q. The results give proof to our theorem.



(a) Elapsed time of AWTM using patterns of length 1 to 128 and weighted text of length 32MB with $k = 4$.



(b) Elapsed time of AWTM using pattern of length 128 and weighted text of length 1MB to 32MB with $k = 4$.



(c) Elapsed time of AWTM using pattern of length 128 and weighted text of length 32MB with k from 4 to 32.

Fig. 2.3 Elapsed time of AWTM with $z = 16$ and $\delta = 4\%$ using synthetic DNA data.

2.4 Experimental results compared with other algorithms and applications in real data

Comparison between algorithms In this paragraph we compared our implementations against other algorithms to evaluate the efficiency. For the first problem WEIGHTEDPATTERNMATCHING, we compared algorithm WPM against the worst-case $\mathcal{O}(nz^2 \log z)$-time algorithm of [13], denoted by WPT and the worst-case $\mathcal{O}(n \log z)$-time algorithm of [44], denoted by KPR. We used synthetic DNA data: where a weighted pattern of length 128, ten texts of length 1MB to 32MB with $1/z = 1/16$. The results were plotted in Fig 2.4 using logarithm scale. It becomes evident from the results that algorithm WPM is one or two orders of magnitude faster than the best worst-case approaches.

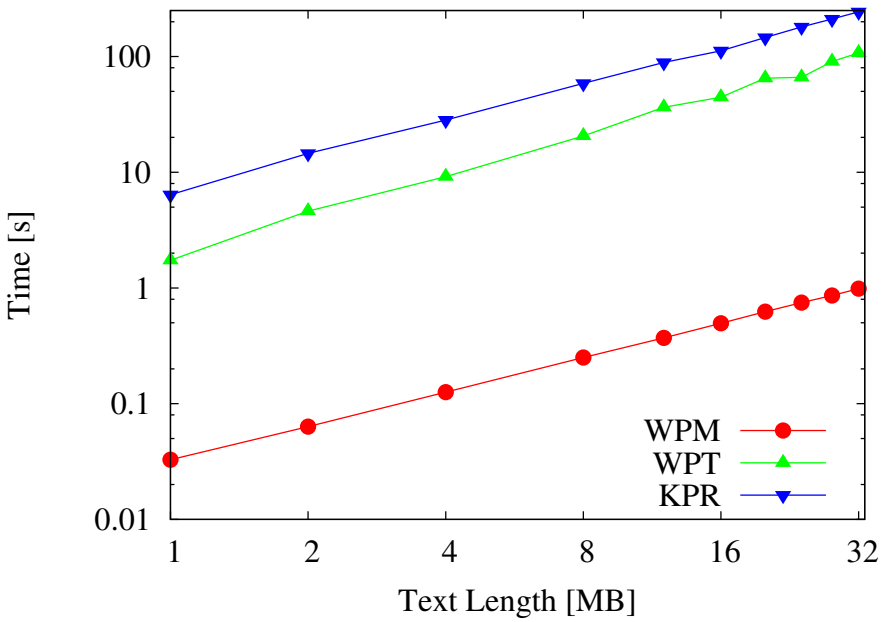
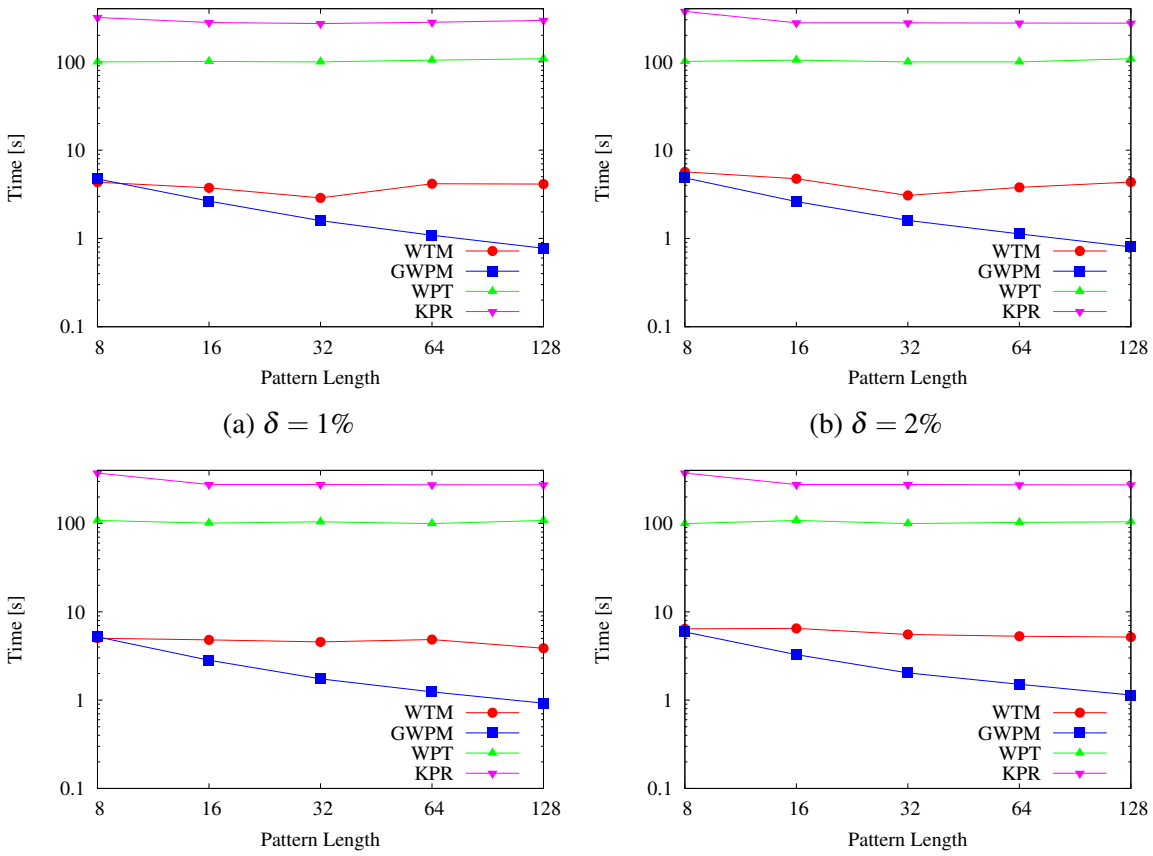


Fig. 2.4 Elapsed time of WPM, WPT and KPR using synthetic DNA data for pattern length 128 and text length 1MB to 32MB with $1/z = 1/16$.

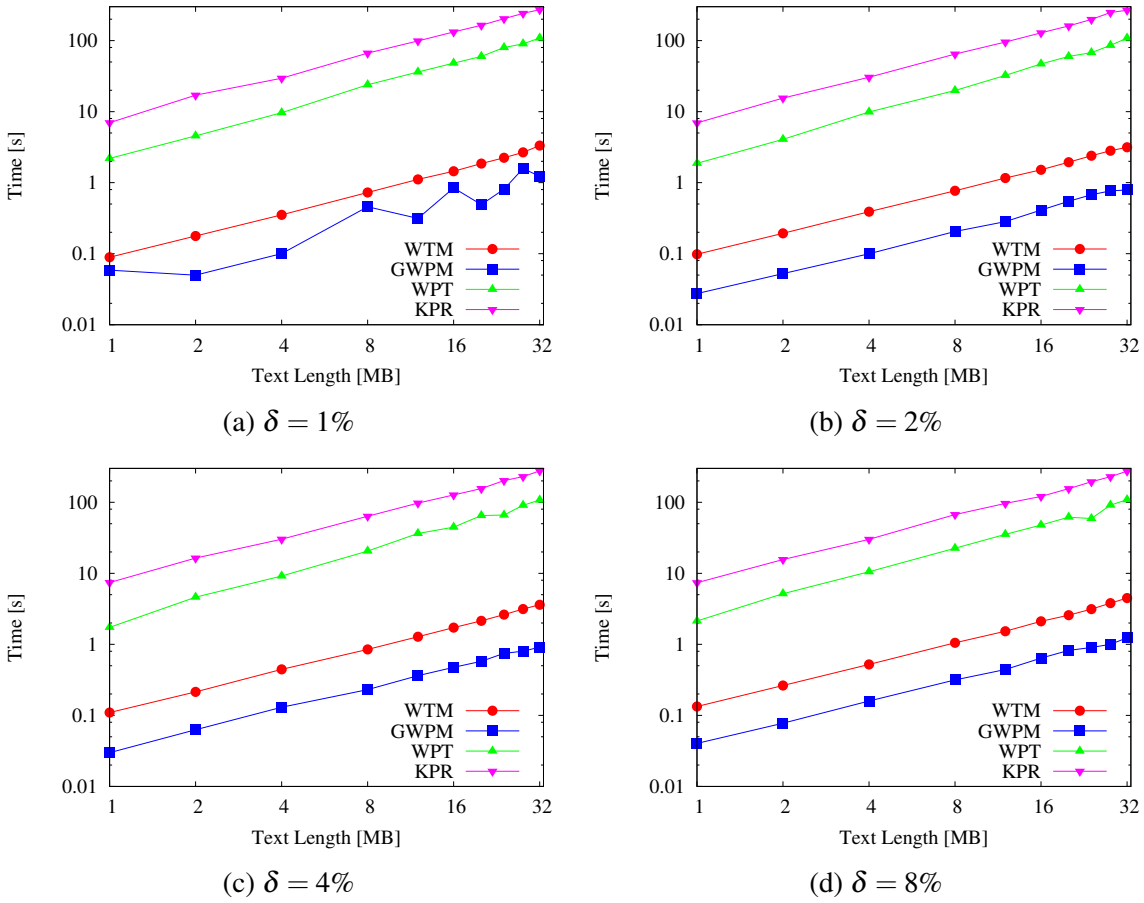
For the second problem WEIGHTEDTEXTMATCHING, we compared our algorithm WTM and GWPM against WPT and KPR. In the first experiment the input pattern we used was a string of length 128, and the input texts were sets of weighted texts of length 1MB to 32MB, with uncertain position percentage δ to be $1\%, 2\%, 4\%, 8\%$ respectively. In the second experiment the input patterns were strings of length 8 to 128, 4 weighted texts of length 32MB with uncertain position percentage δ to be $1\%, 2\%, 4\%, 8\%$ respectively. The threshold for both experiments was set to $1/z = 1/16$. The results were plotted in Fig 2.5 and Fig 2.6 using logarithm scale. From the results we observe: (i) both WTM and GWPM are between one and two orders of magnitude faster than the worst-case approaches, since both worse-case algorithms need to index the input text which is not process in the average case algorithms; (ii) both algorithms have good time performance with short pattern, however GWPM is faster when input pattern is long. This result follows the Theorem 2.3.4.



(a) $\delta = 1\%$ (b) $\delta = 2\%$

(c) $\delta = 4\%$ (d) $\delta = 8\%$

Fig. 2.5 Elapsed time of algorithms WPT, KPR, BLP, and WTM using synthetic DNA data ($\sigma = 4$) for pattern length 128 and weighted text length 1MB to 32MB with $1/z = 1/16$.



(a) $\delta = 1\%$ (b) $\delta = 2\%$

(c) $\delta = 4\%$ (d) $\delta = 8\%$

Fig. 2.6 Elapsed time of algorithms WPT, KPR, BLP, and WTM using synthetic DNA data ($\sigma = 4$) for pattern length 128 and weighted text length 1MB to 32MB with $1/z = 1/16$.

In order to test how the algorithms perform in solving biological problem, we applied two applications using real biological data such as chromosome sequences.

Application I. DnaA proteins are the universal initiators of replication from the chromosomal replication origin (oriC) in bacteria. The DnaA protein recognises and binds specifically to the IUPAC-encoded sequence `TTWTNCACA`, named DnaA box, which is present in all studied bacterial chromosomal replication origins [18]. We transformed sequence `TTWTNCACA` into the weighted sequence `TT[(A,0.5),(T,0.5)]T[(A,0.25),(T,0.25), (G,0.25),(T,0.25)]CACA`;

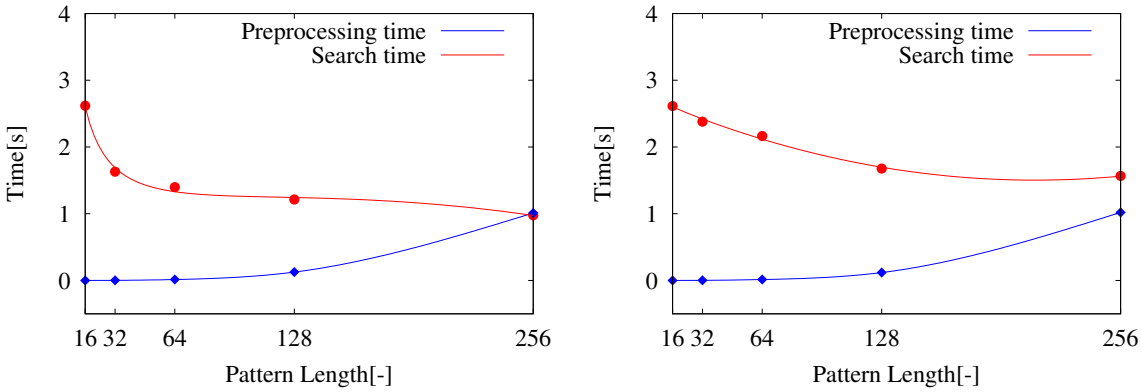
Bacteria	DNA Length	Number of Occurrences	Elapsed Time (s)	
			GWPM	WPT
Bacillus subtilis	4,215,606	321	1.19	9.83
Escherichia coli	4,641,652	165	1.27	10.45
Haemophilus influenzae	1,830,138	177	0.56	4.83
Helicobacter pylori	1,667,867	172	0.48	3.96
Mycobacterium tuberculosis	4,411,532	21	1.14	9.54
Proteus mirabilis	4,063,606	294	1.16	9.23
Pseudomonas aeruginosa	6,264,404	66	1.60	14.66
Pseudomonas putida	6,181,873	141	1.62	13.55
Salmonella enterica	4,857,432	190	1.31	12.03
Staphylococcus aureus	2,821,361	202	0.80	7.75
Streptomyces lividans	8,345,283	18	2.08	18.42
Yersinia pestis	4,653,728	190	1.29	10.72

Table 2.1 Elapsed time of GWPM and WPT for searching for the DnaA box `TTWTNCACA` in 12 bacterial genomes

and then we searched for this sequence in a dozen of bacterial genomes obtained from the NCBI genome database. The length of q-gram was set to $q = 4$ and the cumulative weight threshold to $1/z = 1/8$ to ensure all factors of length 9 are valid. We compared the time performance of algorithm GWPM against the worst-case $\mathcal{O}(nz^2 \log z)$-time algorithm of [13], denoted by WPT, for this assignment. The bacteria used, the number of occurrences found, and the elapsed times are shown in Table 2.1. The results show that GWPM is one order of magnitude faster than WPT and working well in real data.

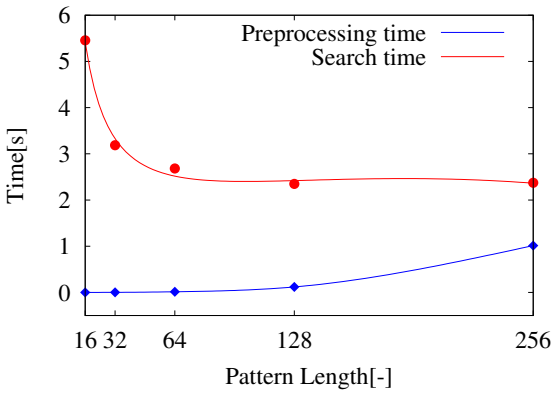
Application II. Mutations in 14 known genes on human chromosome 21 have been identified as the causes of monogenic disorders. These include one form of Alzheimer's disease, amyotrophic lateral sclerosis, autoimmune polyglandular disease, homocystinuria, and progressive myoclonus epilepsy; in addition, a locus for predisposition to leukaemia has been

mapped to chromosome 21 [33]. To this end, we evaluated the time performance of AWTM for pattern matching in a genomes population. Pattern matching is useful for evaluating whether SNPs occur in experimentally derived transcription factor binding sites [31, 59]. As input text we used the human chromosome 21 augmented with a set of genomic variants obtained from the 1000 Genomes Project. The SNPs present in the population were incorporated to transform the chromosome sequence into a weighted text. The length of human chromosome 21 is $48,129,895$ base pairs. The input patterns were selected randomly from the text; their length ranged between 16 and 256. In this real scenario, δ was found to be 0.7%; we therefore set the cumulative weight threshold to the constant value of $1/z = 1/1,024$. For a pattern of length m, the maximum allowed number of mismatches k was set to $2.5\% \times m$, $5\% \times m$, $7.5\% \times m$, and $10\% \times m$. The length of q-grams was set to $q = \log_2 m$ and the number of q-grams read backwards was set to $\ell = \lfloor \frac{k}{0.2 \times q} + 1 \rfloor$. The exact values for q, k, and ℓ are presented in Table 2.2. The results plotted in Fig. 2.7 demonstrate the effectiveness of AWTM: all pattern occurrences can be found within a few seconds, even for error rates of 10%. In addition, our theoretical findings in Theorem 2.3.8 are confirmed: for increasing m and constant n, k, and z, the preprocessing time of AWTM increases but the search time decreases.

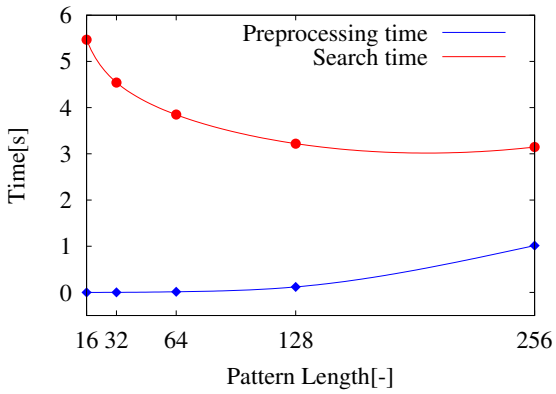


(a) $k = 2.5\% \times m$ and $z = 1,024$

(b) $k = 5.0\% \times m$ and $z = 1,024$



(c) $k = 7.5\% \times m$ and $z = 1,024$



(d) $k = 10\% \times m$ and $z = 1,024$

Fig. 2.7 Elapsed time of AWTM for on-line approximate pattern matching in human chromosome 21 augmented with SNPs from the 1000 Genomes Project.

m	q	2.5%		5.0%		7.5%		10%	
		k	ℓ	k	ℓ	k	ℓ	k	ℓ
16	4	1	2	1	2	2	3	2	3
32	5	1	2	2	3	3	4	4	5
64	6	2	2	4	4	5	5	7	6
128	7	4	3	7	6	10	8	13	10
256	8	7	5	13	9	20	13	26	17

Table 2.2 Pattern length m, q-grams length q, number k of maximum allowed mismatches, and number ℓ of q-grams read backwards for different error rates

Chapter 3

Weighted Indexing

3.1 Introduction

In this chapter, we consider the indexing (or off-line) version of the weighted string matching problem, called *Weighted Indexing*. Here we are given a text being a weighted sequence and we are asked to construct a data structure (called an *index*) to provide efficient operations for answering WPM queries related to the text. We also consider other variants of the indexing problem. In the *Approximate Weighted Indexing* problem, given a pattern and a threshold z', we are to report all occurrences of the pattern with probability at least $1/z'$ but we may also report additional occurrences with probability $1/z' - \varepsilon$, for a pre-selected value of $\varepsilon > 0$. In the *Generalised Weighted Indexing* problem, we are to construct a data structure that allows for WPM queries to be answered for any threshold z' with $z' \leq z$.

A problem that is known to be closely related to the Weighted Indexing problem is *Property Indexing*, first introduced in [7]. In this problem, we are given a string s called

the text and a *hereditary property* Π, which is a family of integer intervals contained in $\{1,\ldots,|s|\}$ (hereditary means that it is closed under subintervals). Our goal is to pre-process the text so that, for a query string p, we can report all occurrences of p in s which, interpreted as intervals, belong to Π. The property Π can be represented in $\mathcal{O}(|s|)$ space using an array $\pi[1\,..\,|s|]$ such the longest interval starting at position i is $\{i,\ldots,\pi[i]\}$.

In each of the indexing problems, we denote the length of the text by n, the length of a query pattern by m, and the number of occurrences of the pattern in the text by Occ.

Previous Result The Weighted Indexing problem was first considered by Iliopoulos et al. [37], who introduced a data structure called *weighted suffix tree* allowing optimal $\mathcal{O}(m+Occ)$-time queries. The construction time and size of that data structure was, however, $\mathcal{O}(n|\Sigma|^{z\log z})$. Their data structure is a compact trie of all of the factors with the probability of occurrence greater than or equal to $1/z$.

Amir et al. [7] reduced the Weighted Indexing problem to the Property Indexing problem in a text of length $\mathcal{O}(nz^2\log z)$. For the latter, they proposed a solution with $\mathcal{O}(n\log\log n)$ preprocessing time and optimal $\mathcal{O}(m+Occ)$ query time. Later, it was shown that the Property Indexing problem can be solved in linear time; see [40, 41, 46]. This led to a solution to the Weighted Indexing problem with index size and construction time $\mathcal{O}(nz^2\log z)$, preserving optimal query time.

Biswas et al. [17] presented a data structure that solves the Approximate Weighted Indexing problem in $\mathcal{O}(\frac{1}{\varepsilon}nz^2)$ space (with $\Omega(\frac{1}{\varepsilon}n^2z^2)$ construction time) with $\mathcal{O}(m+Occ)$-time queries; here Occ denotes the number of occurrences reported. They also proposed a

data structure for the Generalised Weighted Indexing problem with $\mathcal{O}(nz^2 \log z)$ space and $\mathcal{O}(m + m \cdot Occ)$ query time. The construction time is not mentioned, but a direct construction of their index works in $\Omega(n^2 z^2)$ time. Moreover, they also consider the problem of document listing for weighted sequences.

Our Contribution In this chapter, we present two new $\mathcal{O}(nz)$-time construction of $\mathcal{O}(nz)$-sized data structures for the Weighted Indexing problem that answers queries in the optimal $\mathcal{O}(m + Occ)$ time. Our indexes are based on a novel observation that one can always construct a family of $\lfloor z \rfloor$ strings of length n that carries all the information about all the strings that occur in the weighted sequence. This yields a significantly simpler construction than in the previous index [12] preserving all of its applications. As a by-product, we obtain an optimal solution to the Property Suffix Tree problem that avoids complex tools used in the previous solutions [7, 40, 41, 46], and in the Property Suffix Array problem, we provide three $\mathcal{O}(n)$ space worst-case algorithms with time complexity $\mathcal{O}(n \log^2 n)$, $\mathcal{O}(n \log n)$ and $\mathcal{O}(n)$, and an average-case algorithm with time and space $\mathcal{O}(n)$.

We provide proof-of-concept implementations of our weighted indexes that were validated for correctness and efficiency, and present experiments to prove our theorems.

Our approach lets us significantly improve upon the variants of the weighted index proposed in [17].

3.2 z-estimation of Weighted Sequences

In this section we present the construction of a family of $\lfloor z \rfloor$ strings with property from a given weighted sequence. Given a string s on an alphabet Σ, a *property* Π of s is a hereditary collection of integer intervals contained in $\{1,\ldots,n\}$. For simplicity, we represent every property Π with an array $\pi[1\,..\,|s|]$ such that the longest interval $I \in \Pi$ starting at position i is $\{i,\ldots,\pi[i]\}$. Observe that π can be an arbitrary array satisfying $\pi[i] \in \{i-1,\ldots,n\}$ and $\pi[1] \leq \pi[2] \leq \cdots \leq \pi[n]$. For a string p, by $Occ_\pi(p,s)$ we denote the set of occurrences i of p in S such that $i+|p|-1 \leq \pi[i]$.

Let us consider an indexed family $\mathcal{S} = (s_j,\pi_j)_{j=1}^k$ of strings s_j with properties π_j. For a string p and an index i, by

$$Count_{\mathcal{S}}(p,i) = |\{j : i \in Occ_{\pi_j}(p,s_j)\}|$$

we denote the total number of occurrences of p at the position i in the strings $s_1,\ldots,s_k$ that respect the properties.

Our model of computations. We assume the word-RAM model with word size $w = \Omega(\log(nz))$. We consider the log-probability model of representations of weighted sequences in which probabilities can be multiplied exactly in $\mathcal{O}(1)$ time. Without a loss of generality, we further assume that each position contains at most $\lceil z \rceil$ characters with non-zero probability. This is because characters c with $p_i^{(X)} < 1/z$ can be merged into a dummy character $\$_i$.

Consequently, a weighted sequence of length n has a representation using $\mathcal{O}(nz)$ space. We further assume that $\Sigma \subseteq \{1,\ldots,\mathcal{O}(nz)\}$ consists of positive integers.

3.2.1 Existence of an Equivalent Family of Strings

In the definition below, we formalise the property of a family that we aim to construct.

Definition 3.2.1. We say that an indexed family $\mathcal{S} = (s_j, \pi_j)_{j=1}^{\lfloor z \rfloor}$ containing strings s_j of length n is a *z-estimation* of a weighted sequence X of length n if and only if, for every string p and position $i \in \{1,\ldots,n\}$, $\mathit{Count}_{\mathcal{S}}(p,i) = \left\lfloor \Pi_X(p,i) z \right\rfloor$, where $\Pi_X(p,i)$ denotes the cumulative probability of p beginning at position i in X.

Note that a z-estimation $\mathcal{S}$ of a weighted sequence X carries the information about all solid factors of X: a string p occurs in X at position i if and only if it occurs at position i in at least one of the strings S_j respecting its property π_j. This observation will be used in the construction of our weighted index. Moreover, the value $\mathit{Count}_{\mathcal{S}}(p,i)$ provides a good estimation of the probability $\Pi_X(p,i)$:

$$\tfrac{1}{z}\mathit{Count}_{\mathcal{S}}(p,i) \leq \Pi_X(p,i) < \tfrac{1}{z}\mathit{Count}_{\mathcal{S}}(p,i) + \tfrac{1}{z}.$$

This will let us design an approximate weighted index. An example of a z-estimation of weighted string

$$X = \mathtt{a}[(\mathtt{a},0.5),(\mathtt{b},0.5)][(\mathtt{a},0.75),(\mathtt{b},0.25)][(\mathtt{a},0.8),(\mathtt{b},0.2)][(\mathtt{a},0.5),(\mathtt{b},0.5)][(\mathtt{a},0.25),(\mathtt{b},0.75)] \tag{3.1}$$

is shown in Table 3.1 (running example).

i	0	1	2	3	4	5
$S_1[i]$	a	a	a	a	a	a
$\pi_1[i]$	2	2	3	4	5	6
$S_2[i]$	a	a	a	a	a	b
$\pi_2[i]$	4	4	5	6	6	6
$S_3[i]$	a	b	a	a	b	b
$\pi_3[i]$	4	4	5	6	6	6
$S_4[i]$	a	b	b	b	b	b
$\pi_4[i]$	2	2	3	3	5	6

(a) A 4-estimation $\mathcal{S}$ of weighted sequence X.

string P	$\Pi_X(P,2)$	$\{j : 2 \in Occ_{\pi_j}(P,S_j)\}$
ε	1	1, 2, 3, 4
a	0.75	1, 2, 3
aa	0.6	2, 3
aaa	0.3	2
aab	0.3	3
b	0.25	4

(b) All the strings that occur at position $i = 2$ in X together with the probabilities of occurrence in X and occurrences in $\mathcal{S}$.

Table 3.1 Running example of weighted string X in (3.1).

Below, we prove the existence of a z-estimation. An efficient construction is deferred to the next section.

For a fixed weighted sequence X of length n and a threshold z, we can use compact notation:

$$t_i(p) = \left\lfloor \Pi_X(p,i)z \right\rfloor \quad \text{and} \quad m_i(p) = t_i(p) - \sum_{c\in\Sigma} t_i(p\alpha)$$

for $i = 1,\ldots,n$. (Note that $p\alpha$ denotes the concatenation of p and the letter α.) We start with an equivalent characterisation of z-estimations of X.

Observation 3.2.2. *A family $\mathcal{S} = (s_j,\pi_j)_{j=1}^{\lfloor z\rfloor}$ is a z-estimation of X if and only if for each position i, every string p is a prefix of exactly $t_i(p)$ strings $s_j[i\,..\,\pi_j[i]]$.*

Next, we prove that this condition uniquely defines the multi-set

$$\{s_j[i\,..\,\pi_j[i]] : 1 \le j \le \lfloor z\rfloor\}.$$

Lemma 3.2.3. *There exists a unique multi-set $\mathcal{M}_i$ such that each string p is a prefix of exactly $t_i(p)$ strings in $\mathcal{M}_i$.*

Proof. Consider a multi-set $\mathcal{M}_i$ satisfying the required condition and an arbitrary string p. For each $\alpha\in\Sigma$, there are $t_i(p\alpha)$ strings in $\mathcal{M}_i$ with the prefix p followed by a letter α. In the remaining $t_i(p) - \sum_{\alpha\in\Sigma} t_i(p\alpha)$ strings in $\mathcal{M}_i$, the prefix p is not followed by any letter. Thus, the multiplicity of p in $\mathcal{M}_i$ must be $m_i(p)$. This implies uniqueness of $\mathcal{M}_i$.

Note that $t_i(p) = \lfloor \Pi_X(p,i)z \rfloor \geq \sum_{\alpha\in\Sigma} t_i(p\alpha)$ because $\Pi_X(p,i) \geq \sum_{\alpha\in\Sigma} \Pi_X(p\alpha,i)$ and the function $f(x) = \lfloor c \cdot x \rfloor$ is super-additive where c is a constant value(a function $f(x)$ is super-additive if $f(x+y) \geq f(x)+f(y)$ for all x and y). Consequently, we may define a multi-set $\mathcal{M}_i$ using values $m_i(p)$ as multiplicities. It remains to prove that this multi-set satisfies the required condition. For this, we consider strings p in the order of decreasing lengths. The base case is trivial because strings P longer than X satisfy $\Pi_X(p,i) = 0$. The inductive hypothesis yields that, for each $c \in \Sigma$, the string $p\alpha$ is a prefix of $t_i(p\alpha)$ strings in $\mathcal{M}_i$. Consequently, the string p is a prefix of $m_i(p) + \sum_{\alpha\in\Sigma} t_i(p\alpha) = t_i(p)$ strings in $\mathcal{M}_i$, as claimed. □

Observe that in a z-estimation, $S_j[i\,..\,\pi_j[i]]$ can be obtained from $S_j[i+1\,..\,\pi_j[i+1]]$ by inserting a leading letter and dropping some number of trailing letters. (This includes dropping the newly inserted letter if $S_j[i\,..\,\pi_j[i]] = \varepsilon$.) The relation between these strings can be formalised as follows:

Definition 3.2.4. We say that $p \in \mathcal{M}_i$ is *compatible* with $q \in \mathcal{M}_{i+1}$ if $p = \varepsilon$ or $p = \alpha q'$ for some letter $\alpha \in \Sigma$ and a prefix q' of q.

Thus, if a z-estimation exists, it yields a perfect matching between $\mathcal{M}_{i+1}$ and $\mathcal{M}_i$ such that the matched strings are compatible. We prove that such a matching exists unconditionally. For an example, see Example 3.2.1.

Example 3.2.1. A running example of sets $\mathcal{M}_i$ for the weighted sequence X in (3.1) with $z = 4$ is shown in following table. Perfect matchings of compatible strings between $\mathcal{M}_i$ and $\mathcal{M}_{i+1}$ are marked. The first letters of the strings form the 4-estimation from Table 3.1 and

the length of the j-th string in $\mathcal{M}_i$ corresponds to $\pi_j[i]-i$. Note that if we consider $\mathcal{M}_2$ and string aa, we have $t_2(\mathtt{aa}) = \lfloor 0.75 \times 0.8 \times 4) \rfloor = 2$, and we have exact 2 strings that have prefix aa in $\mathcal{M}_2$.

$\mathcal{M}_0$		$\mathcal{M}_1$		$\mathcal{M}_2$		$\mathcal{M}_3$		$\mathcal{M}_4$		$\mathcal{M}_5$
aa	—	a	—	a	—	a	—	a	—	a
aaaa	—	aaa	—	aaa	—	aab	—	ab	—	b
abaa	—	baa	—	aab	—	abb	—	bb	—	b
ab	—	b	—	b	—	ε	—	b	—	b

Lemma 3.2.5. *For every $0 \leq i < n-1$, there exists a one-to-one correspondence from $\mathcal{M}_{i+1}$ into $\mathcal{M}_i$ such that each $q \in \mathcal{M}_{i+1}$ is matched with a compatible $p \in \mathcal{M}_i$.*

Proof. We greedily transform each $q \in \mathcal{M}_{i+1}$ into the longest compatible $p \in \mathcal{M}_i$ which is still unmatched. If no compatible $p \in \mathcal{M}_i$ is available, we leave q unmatched. We will show that all strings $q \in \mathcal{M}_{i+1}$ are actually matched at the end of this process. Since $|\mathcal{M}_i| = t_i(\varepsilon) = \lfloor z \rfloor = t_{i+1}(\varepsilon) = |\mathcal{M}_{i+1}|$, it suffices to prove that no $p \in \mathcal{M}_i$ is left unmatched.

An empty string $p \in \mathcal{M}_i$ is compatible with every $q \in \mathcal{M}_{i+1}$, so it cannot be left unmatched. Thus, suppose that $p = \alpha q' \in \mathcal{M}_i$, for some $\alpha \in \Sigma$ and string q', is left unmatched. Let us denote by $\mathcal{R}$ the multi-set containing all strings $q \in \mathcal{M}_{i+1}$ compatible with p, i.e., starting with q'. We further define $\mathcal{L}$ as the multi-set containing all strings $p' \in \mathcal{M}_i$ that start with $\alpha' q'$ for some $\alpha' \in \Sigma$. The construction procedure guarantees that each $q \in \mathcal{R}$ has been matched to a compatible p' satisfying $|p'| \geq |p|$; such p' must belong to the multiset $\mathcal{L}$.

Observe that $|\mathcal{L}| = \sum_{\alpha' \in \Sigma} t_i(\alpha' q') \leq t_{i+1}(q') = |\mathcal{R}|$ because

$$\Pi_X(q', i+1) = \sum_{\alpha' \in \Sigma} \Pi_X(\alpha' q', i)$$

and the function $f(x) = \lfloor xz \rfloor$ is super-additive. Consequently, each $p' \in \mathcal{L}$ must be matched to some $q \in \mathcal{R}$. Since $p \in \mathcal{L}$ is unmatched, we obtain a contradiction. □

Due to Lemma 3.2.5, we can index the strings $\mathcal{M}_i = \{p_{j,i} : 1 \leq j \leq \lfloor z \rfloor\}$ so that we have $\lfloor z \rfloor$ chains $p_{j,1}, \ldots, p_{j,n}, p_{j,n+1} = \varepsilon$ with compatible subsequent strings. It is easy to transform each such chain to a string s_j with property π_j so that $s_j[i..\pi_j[i]] = p_{j,i}$. The value $s_j[i]$ is not specified if $p_{j,i} = \varepsilon$; in this case, we may set $s_j[i]$ to an arbitrary letter. The resulting family $\mathcal{S} = (s_j, \pi_j)_{j=1}^{\lfloor z \rfloor}$ clearly satisfies the characterisation of Observation 3.2.2, which completes the proof of the following result.

Theorem 3.2.6. *Each weighted sequence has a z-estimation.*

3.2.2 Efficient Implementation

In this section, we describe an algorithm which, given a weighted sequence X of length n and a threshold z, constructs a z-estimation of X in $\mathcal{O}(nz)$ time.

At a high level, we follow the existential construction of Section 3.2.1. We start with $\mathcal{M}_{n+1}$, which consists of $\lfloor z \rfloor$ copies of ε, and we iterate over positions $i = n, \ldots, 1$ transforming $\mathcal{M}_{i+1}$ to $\mathcal{M}_i$ so that each $p_{j,i+1} \in \mathcal{M}_{i+1}$ is replaced with a compatible string $p_{j,i} \in \mathcal{M}_i$. We simultaneously build the z-estimation $\mathcal{S} = (s_j, \pi_j)_{j=1}^{\lfloor z \rfloor}$. More precisely, we set $\pi_j[i]$ to $i + |p_{j,i}| - 1$ and $s_j[i]$ to the leading letter of $p_{j,i}$, or an arbitrary letter if $p_{j,i} = \varepsilon$.

Each transformation follows the lines of the procedure described in the proof of Lemma 3.2.5. However, our implementation uses *solid factor tries* in order to achieve $\mathcal{O}(z)$ amortised running time. We introduce this auxiliary data structure below.

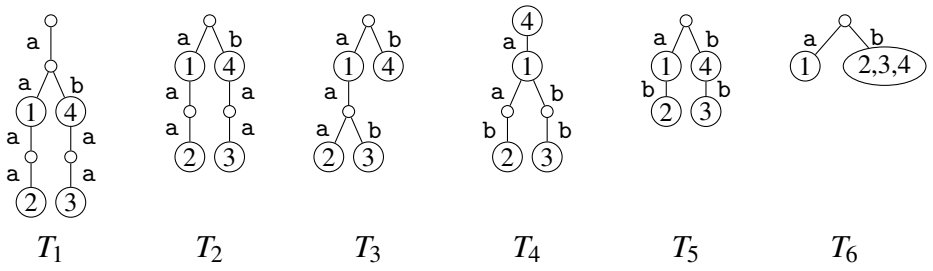


Fig. 3.1 The solid factor tries for the weighted sequence X in (3.1) with $z = 4$. Tokens in the nodes are numbered according to the order from Table **??**.

Solid Factor Tries

Recall that a trie is a rooted tree in which each node represents a string; the string corresponding to node u, called the *label* of u, is denoted $\mathsf{L}(u)$. The root has label ε, and the parent of a node u with $\mathsf{L}(u) = p\alpha$ for $\alpha \in \Sigma$ is the node v with $\mathsf{L}(v) = p$; the edge from p to $p\alpha$ is *labelled* with α. Observe that the family of solid factors occurring at position i (i.e., strings P such that $t_i(p) > 0$) is closed with respect to prefixes. Thus, we can define a *solid factor trie* T_i whose nodes represent these factors.

We store $\mathcal{M}_i$ using *tokens* in T_i: each $p_{j,i} \in \mathcal{M}_i$ is represented by a token (with identifier j) located at the node $u \in T_i$ with $\mathsf{L}(u) = p_{j,i}$. For each token j, we store the node $u \in T_i$ with $\mathsf{L}(u) = p_{j,i}$ and the probability $\Pi_X(p_{j,i}, i)$. Observe that the number of tokens at the node u is $m_i(\mathsf{L}(u))$ and the number of tokens in the subtree rooted at u is $t_i(\mathsf{L}(u))$. To simplify notation, we denote $m_i(u) = m_i(\mathsf{L}(u))$ and $t_i(u) = t_i(\mathsf{L}(u))$. We have the following simple observation; for an example see Figure 3.1 (running example).

Observation 3.2.7. *The trie T_i contains $\lfloor z \rfloor$ tokens in total and every leaf contains tokens.*

Transformation Algorithm for Alphabets of Constant Size

For each index i, we transform the solid factor trie T_{i+1} to T_i and move the tokens so that $\mathcal{M}_{i+1}$ is transformed to $\mathcal{M}_i$. Before we describe the implementation, let us formulate a relation between T_i and T_{i+1}.

Observation 3.2.8. *If $u \in T_i$ has a non-empty label, $\mathsf{L}(u) = \alpha p$, for some $\alpha \in \Sigma$, then T_{i+1} contains a node v with label $\mathsf{L}(v) = p$.*

Consequently, each non-root node $u \in T_i$ has a corresponding node $v \in T_{i+1}$. In our construction algorithm, we sometimes reuse v as u; otherwise, we create u as a copy of v. More precisely, we distinguish a *heavy letter* $h \in \Sigma$ maximising probability $prob_i^{(X)}(\alpha)$ over $\alpha \in \Sigma$. We reuse v if $\mathsf{L}(u)$ starts with h and create a copy of v otherwise.

This approach is implemented as follows. First, we create the root of T_i and attach T_{i+1} to the new root using an edge with label h. The resulting subtree, denoted $T_{i,h}$, contains all tokens present in T_{i+1} and may contain nodes v with $t_i(v) = 0$ (we piggyback trimming them to the last phase when tokens are moved). Next, we consider all the remaining letters $\alpha \in \Sigma \setminus \{h\}$. For each such letter, we shall build a subtree $T_{i,\alpha}$ representing solid factors occurring at position i and starting with letter α. We simultaneously build and traverse $T_{i,\alpha}$: we construct the children of a node u while visiting u for the first time. While at node u with $\mathsf{L}(u) = \alpha p$, we maintain the probability $\Pi_X(\alpha p, i)$ and a pointer to the corresponding node $v \in T_{i,h}$ such that $\mathsf{L}(v) = hp$. To construct the children of u, we simply compute $t_i(\alpha p \beta)$ for each $\beta \in \Sigma$. Moreover, we determine $m_i(\alpha p)$ and place $m_i(\alpha p)$ *token requests* at node v, announcing that $m_i(\alpha p)$ tokens are needed at u.

Finally, we move the tokens and trim the redundant nodes of $T_{i,h}$. We process the tokens in an arbitrary order. Consider a token located at node v of $T_{i,h}$ with $\mathsf{L}(v) = hq$ (the token used to represent $q \in \mathcal{M}_{i+1}$). We traverse the path from v towards the root of T_i maintaining the probability $\Pi_X(\mathsf{L}(v'), i)$ at the currently visited node v'. First, we check if there is any token request at v'. If so, we comply with the request, remove it, and terminate the traversal. Otherwise, we compute $m_i(v')$ using the probability. If v' contains less than $m_i(v')$ already processed tokens, we place our token at v' and terminate the traversal. Otherwise, we proceed to the parent of v'. If v' is a leaf and does not contain any (processed or unprocessed) tokens, we remove v' from $T_{i,h}$. If the traversal reaches the root of T_i, we place the token unconditionally at the root. For an example, see Figure 3.2 (running example).

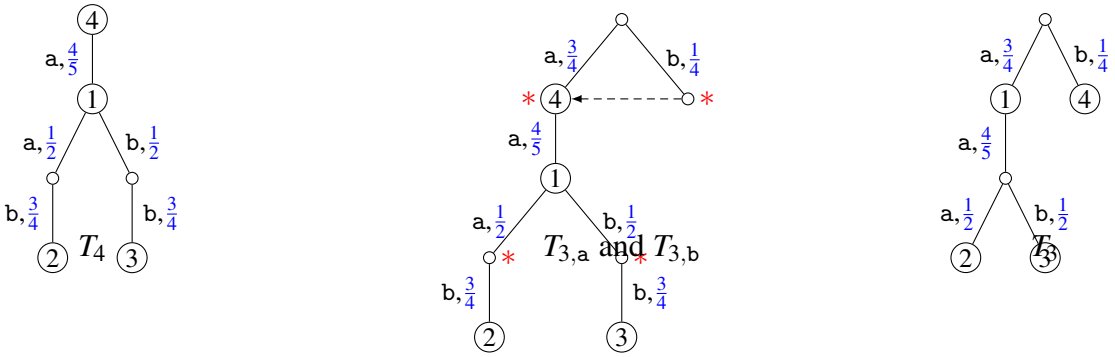


Fig. 3.2 Transformation between T_4 to T_3 from the example in Figure 3.1. To the left: the trie T_4 with letter probabilities (in blue). In the middle: the trie T_4 is copied as $T_{3,\mathsf{a}}$, whereas $T_{3,\mathsf{b}}$ is created using a backtracking algorithm (in this case, it has only one node). Asterisks denote nodes that require tokens. The token request is shown with an arrow. To the right: the final T_3 created after the tokens are moved up and redundant nodes are removed. Note that the tokens number 1 and 4 could have been interchanged depending on the order of processing.

Correctness We shall prove that the procedure described above correctly computes T_i and $\mathcal{M}_i$. Due to Observation 3.2.8, the trie T_i contains all the necessary nodes. We only need

to prove that no redundant nodes v (with $t_i(v) = 0$) are left in $T_{i,h}$. Suppose that v is the deepest such node; clearly, it must be a leaf of $T_{i,h}$. We did not place the token at v because $m_i(v) \leq t_i(v) = 0$. On the other hand, tokens were presented in all leaves of T_{i+1}, so the subtree of v in $T_{i,h}$ initially contained a token. Let us consider the moment of moving the last token in this subtree. If the token travelled further to the parent of v, we would have removed v. Thus, the token must have been placed at a node u complying with a token request at v. However, in that case we have $t_i(v) \geq t_i(u) \geq m_i(u) > 0$, because h is the heavy letter. This contradiction concludes the proof.

Hence, we proceed to prove that the final configuration of tokens represents $\mathcal{M}_i$. For this, we observe that our algorithm simulates the greedy procedure in the proof of Lemma 3.2.5. In other words, we shall prove that we transformed $p_{j,i+1} \in \mathcal{M}_{i+1}$ to the longest compatible element of $\mathcal{M}_i$ which was still unmatched when we processed token j. Suppose that there was an unmatched string $p' \in \mathcal{M}_i$ longer than $p_{j,i}$. Let $p' = \alpha q'$ and observe that, when processing token j, we visited the node v' with $\mathsf{L}(v') = hq'$. If $\alpha = h$, then we would have less than $m_i(v')$ processed tokens at v'. Otherwise, there must have been a token request at v'. In either event, we would not have proceeded to the parent of v'. This contradiction concludes the proof.

Running Time Analysis It remains to show that the total running time of the n transformations is $\mathcal{O}(nz)$. In a single iteration, processing the j-th token, i.e., transforming $P_{j,i+1}$ to $P_{j,i}$, we visited at most $1 + |p_{j,i+1}| - |p_{j,i}|$ nodes of $T_{i,h}$ and deleted some of them. Across all iterations this is $\mathcal{O}(n)$ per token and $\mathcal{O}(nz)$ in total. The remaining operations (construction

of subtrees $T_{i,c}$) take $\mathcal{O}(|\Sigma|) = \mathcal{O}(1)$ time per created node. The final tree T_1 has $\mathcal{O}(nz)$ nodes and the overall number of deleted nodes is $\mathcal{O}(nz)$. Hence, the total number of created nodes is also $\mathcal{O}(nz)$.

This concludes the proof that the running time is $\mathcal{O}(nz)$. Hence, we achieve the main goal of this section.

Theorem 3.2.9. *For a weighted sequence X of length n over a constant-sized alphabet, one can construct a z-estimation in $\mathcal{O}(nz)$ time.*

3.3 Weighted Suffix Tree

In this section, we apply suffix tree on weighted sequences. The main problem can be formulated as follows.

WEIGHTEDSUFFIXTREE

Input: A weighted string X of length n over an alphabet Σ and a cumulative weight threshold $1/z \in (0,1]$

Output: A weighted suffix tree of X such that for a given pattern string p of length m, check if $Occ_{\frac{1}{z}}(p,X) \neq \emptyset$ (*decision* query), compute $|Occ_{\frac{1}{z}}(p,X)|$ (*counting* query), or report all elements of $Occ_{\frac{1}{z}}(p,X)$ (*reporting* query).

Instead of constructing the weighted suffix tree directly, we consider the property suffix tree problem first.

PROPERTYSUFFIXTREE

Input: A string s of length n over an alphabet Σ and an array π representing a property Π.

Output: A property suffix tree of s such that for a given pattern string p of length m,compute $|Occ_\pi(p,s)|$ or report all elements of $Occ_\pi(p,s)$.

3.3.1 Property Suffix Tree Construction

In analogy to the suffix tree, given a string S with property Π represented by an array π, we define the *property suffix tree* of (s,π) as the compact trie representing strings $s[i..\pi[i]]$. Each terminal node v stores a list L_v containing all indices i such that $s[i..\pi[i]]$ is the path-label of v. This way, $Occ_\pi(p,s)$ can be retrieved by locating the locus of p and writing down indices in lists L_v for all descendants of the locus.

For a given string s, we construct the property suffix tree with respect to property Π from the suffix tree of s. This process is implemented in three steps. First, for each index i we determine the locus v_i of $s[i..\pi[i]]$. Next, we make all these loci explicit to create new terminal nodes. Finally, we remove nodes which should no longer exist in the tree or no longer be explicit.

Our approach to the first phase is similar to Ukkonen's suffix tree construction [63]. We are to determine the locus v_i of $s[i..\pi[i]]$. For this, we shall traverse the suffix tree starting from an explicit node u_i guaranteed to be an ancestor of v_i. We obtain u_i by following the suffix link of the nearest explicit ancestor of v_{i-1} (v_{i-1} itself if it is explicit). If $i = 1$ or the explicit ancestor of v_{i-1} is the root, we simply set u_i as the root. Since $\pi[i] \geq \pi[i-1]$ for

$i > 1$, u_i is indeed an ancestor of v_i. Therefore, we can progress down the edges in the suffix tree from u_i, keeping track of the current depth until the desired depth is reached. We know that v_i exists in the tree, so it suffices to read only the first letters of each traversed edge.

This procedure results in the sequence of loci $(v_i)_{i=1}^{n}$. Let us analysis its time complexity.

In the i-th iteration we traverse: one edge to reach u_i, then several edges to a node whose suffix link is u_{i+1}, and finally at most one edge to reach v_i. Hence, the number of edges traversed in this iteration is at most $2+|\mathsf{L}(u')|-|\mathsf{L}(u_i)| \leq 3+|\mathsf{L}(u_{i+1})|-|\mathsf{L}(u_i)|$, which gives $\mathcal{O}(n)$ overall.

The remaining steps of the algorithm are performed as follows. We sort the loci v_i by the path label length $\pi[i]-i+1$ and group them based on the edge where they are located. This allows us appropriately subdivide each edge and compute the lists L_v for the new terminal nodes. Finally, we trim the tree: we traverse the tree bottom-up and remove or dissolve nodes which should no longer be explicit. These steps clearly work in $\mathcal{O}(n)$ time.

3.3.2 Weighted Suffix Tree

Now we are able to describe our data structure for the WEIGHTEDSUFFIXTREE problem. For a weighted sequence X and a threshold z, we construct a z-estimation $\mathcal{S} = (s_j, \pi_j)_{j=1}^{\lfloor z \rfloor}$ of X, concatenate all the strings and shift the properties so that a single string s with property π is obtained. Our weighted index is the property suffix tree of s and π. In the property suffix tree, each terminal node is labelled by the list of all the occurrences of the corresponding string in s respecting its property. We shift these indices so that they describe the indices within the respective strings s_j.

We assume left-to-right orientation of the children of each node (e.g., lexicographic). A global occurrence list *OL* is stored being a concatenation of the lists of occurrences in all the terminal nodes in pre-order. Each node v stores, as $OL(v)$, the occurrence list of terminal nodes in its subtree represented as a pair of pointers to elements of the global list *OL*. We enhance the occurrence list *OL* by a data structure for the following coloured range listing problem.

Problem (Coloured range listing)**.** Preprocess a sequence $A[1..N]$ of elements from $[1..S]$ so that, given a range $A[i..j]$, one can list all the distinct elements in that range.

Fact 3.3.1 (Muthukrishnan [53])**.** *A data structure for the coloured range listing problem of $\mathcal{O}(N)$ size can be constructed in $\mathcal{O}(N+S)$ time and answers queries in $\mathcal{O}(k+1)$ time where k is the number of distinct elements reported.*

For all nodes we also compute the following values (for the purpose of this computation we replace each leaf v with $|OL(v)|$ bogus leaves with single occurrences).

Fact 3.3.2 (Colour set size, Hui [35])**.** *Given a rooted tree of size N with L leaves coloured from $[1..S]$, in $\mathcal{O}(N+S)$ time one can find for each node u the number of distinct leaf colours in the subtree of u.*

The space complexity of the index is obvious $\mathcal{O}(nz)$, where n is the length of X. Theorem 3.2.9 shows that the data structure can be constructed in $\mathcal{O}(nz)$ time. The resulting weighted index is very similar to the one constructed in [12], even though the construction algorithm is very different.

By Definition 3.2.1, a string p occurs at position i in X if and only if it occurs at this position in at least one of the strings. Thus, to check if $Occ_{\frac{1}{z}}(p,X) \neq \emptyset$, it suffices to traverse down the property suffix tree and check if it contains a node v corresponding to p. This search takes $\mathcal{O}(m)$ time, where $m = |p|$. We can use Fact 3.3.2 to equip each explicit node with the number of positions where the string represented by the node occurs. This way, $|Occ_{\frac{1}{z}}(p,X)|$ can also be determined in $\mathcal{O}(m)$ time. With the aid of the data structure for coloured range listing, we can also report $Occ_{\frac{1}{z}}(p,X)$ in time proportional to the number of reported elements. We thus obtain the following result.

Theorem 3.3.3. *For a weighted sequence X of length n over an integer alphabet and a threshold z, there is a weighted index of $\mathcal{O}(nz)$ size that can be constructed in $\mathcal{O}(nz)$ time and answers decision and counting queries in $\mathcal{O}(m)$ time and reporting queries in $\mathcal{O}(m+|Occ_{\frac{1}{z}}(p,X)|)$ time.*

3.3.3 Approximate Weighted Index

Let us proceed to the solution of the Approximate Weighted Indexing problem. We are to answer queries for a pattern p and a probability threshold $1/z'$ and are allowed to report occurrences with probability $\geq 1/z' - \varepsilon$, for a given value of $\varepsilon > 0$. Let us recall that for constant-sized alphabets [17] solves this problem in $\mathcal{O}(\frac{1}{\varepsilon}nz^2)$ space (with $\Omega(\frac{1}{\varepsilon}n^2z^2)$ construction time) with $\mathcal{O}(m+|Occ_{1/z'-\varepsilon}(P,X)|)$-time queries, assuming that $z' \leq z$ holds in all queries. Our techniques lead to a substantial improvement over the complexities of this index.

Assume that the query is for a pattern p and a threshold $1/z'$. If $1/z' < \varepsilon$, then the query is trivial as all the positions in X can be reported. Henceforth, we assume that $1/z' \geq \varepsilon$.

Let us consider a z-estimation $\mathcal{S}$ for the weighted sequence with $z = 1/\varepsilon$. Let $\ell = \left\lfloor \frac{z}{z'} \right\rfloor$. By Definition 3.2.1, we can return position i as an occurrence of p based on whether $Count_{\mathcal{S}}(p,i) \geq \ell$; this is shown in the following lemma.

Lemma 3.3.4. *If $Count_{\mathcal{S}}(p,i) \geq \ell$, then $\Pi_X(p,i) \geq 1/z' - \varepsilon$. If $Count_{\mathcal{S}}(p,i) < \ell$, then $\Pi_X(p,i) < 1/z'$.*

Proof. Assume that $Count_{\mathcal{S}}(p,i) \geq \ell$. Then

$$\Pi_X(p,i) \geq \tfrac{1}{z} Count_{\mathcal{S}}(p,i) \geq \tfrac{1}{z}\left\lfloor \tfrac{z}{z'} \right\rfloor \geq \tfrac{1}{z}(\tfrac{z}{z'} - 1) = \tfrac{1}{z'} - \varepsilon.$$

Now assume that $Count_{\mathcal{S}}(p,i) < \ell$. As $Count_{\mathcal{S}}(p,i) = \lfloor \Pi_X(p,i) z \rfloor$, this concludes that $\Pi_X(p,i) z < \ell$, which is equivalent to $\Pi_X(p,i) < \frac{\ell}{z} = \frac{1}{z}\left\lfloor \frac{z}{z'} \right\rfloor \leq \frac{1}{z'}$. □

Thus our approximate weighted index for X is the weighted index for X constructed for $z = 1/\varepsilon$. To obtain the desired accuracy, it suffices to find the node v in the property suffix tree that corresponds to p and report all positions i in X such that there are at least $\left\lfloor \frac{z}{z'} \right\rfloor$ leaves in the subtree of v labelled with the position i. Let us show that this can be done by augmenting the weighted index by a data structure for *(top-k) document retrieval*.

A version of the document retrieval problem (see Section 4.1 in [55]) can be stated operationally as follows. We are given a compact trie T with N leaves, each leaf labelled with a document number being a positive integer up to N. (Usually, T is a suffix tree of a collection of documents.) Given a pattern p, let v be the locus of p. Our goal is to report

subsequent documents whose numbers occur most frequently in the leaves of the subtree of v until the process of reporting is interrupted. In [55] a data structure of size $\mathcal{O}(N)$ is shown that, given the node v, reports k top-scoring documents in $\mathcal{O}(k)$ time. The construction time of the data structure is $\mathcal{O}(N \log N)$.

We can augment our property suffix tree with this data structure with the document numbers being the labels of terminals (we can create a separate leaf for each label). This gives $N = \mathcal{O}(nz) = \mathcal{O}(\frac{n}{\varepsilon})$. To find the documents with at least ℓ occurrences, we compute by doubling the smallest k such that the last of the top k documents reported has less than ℓ occurrences. The number of documents reported in the last step of the doubling search will be at most $2|Occ_{\frac{1}{z'}-\varepsilon}(p,X)|$ and the total number will not exceed $4|Occ_{\frac{1}{z'}-\varepsilon}(p,X)|$.

Theorem 3.3.5. *For a weighted sequence of length n over an integer alphabet and parameter $\varepsilon > 0$, the Approximate Weighted Indexing problem can be solved in $\mathcal{O}(\frac{n}{\varepsilon})$ space with $\mathcal{O}(m + |Occ_{\frac{1}{z'}-\varepsilon}(P,X)|)$-time queries. The construction time is $\mathcal{O}(\frac{n}{\varepsilon}\log\frac{n}{\varepsilon})$.*

3.4 Weighted Suffix Array

In this section, we present the weighted suffix array problem. Similar with last section, we present the construction of property suffix array first.

3.4.1 Introduction

Recall that the *suffix array* (SA) of a text x of length n is an integer array of size n that stores the lexicographically sorted list of suffixes of x [49]. In order to construct the *Property Suffix*

Array, which is denoted by PSA, we essentially need to lexicographically sort a multiset consisting of substrings of x; this multiset contains at most one prefix of each suffix of x. This can be achieved in linear time by traversing the PST, however our aim here is to do it directly—we do not want to construct or store the PST. It is well-known from the setting of standard strings that the SA is more space efficient than the suffix tree [2].

Note that for clarity of presentation we represent Π—and assume the input is given in this form—by an integer array $\mathcal{L}$ of size n, such that

$$\mathcal{L}[i] = \max\{j | (k,j) \in \Pi, k \leq i\} - i + 1$$

is the length of the longest prefix of $x[i..n-1]$ that is valid. It should be clear that $\mathcal{L}$ can be obtained from Π in $\mathcal{O}(n+|\Pi|)$ time. We also assume that $\mathcal{L}[i] > 0$ for all i; the case that $\mathcal{L}[i] = 0$ can be handled easily as the resulting substring would just be the empty string.

Example 3.4.1 (Running example)**.** Consider the string $x = \texttt{acababaab}$ and property $\Pi = \{(0,3),(4,6),(6,8)\}$:

i	0	1	2	3	4	5	6	7	8
$x[i]$	a	c	a	b	a	b	a	a	b
$\mathcal{L}[i]$	4	3	2	1	3	2	3	2	1
SA$[i]$	6	7	4	2	0	8	5	3	1
PSA$[i]$	6	2	7	4	0	3	8	5	1

Our main result is an $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space direct construction of the PSA for integer alphabets. The problem can be formally defined as follows.

PROPERTY SUFFIX ARRAY

Input: A string x of length n and an integer array $\mathcal{L}$ of size n, satisfying $0 < \mathcal{L}[i] \leq n-i$ and $\mathcal{L}[i] \geq \mathcal{L}[i-1]-1$.

Output: An array PSA that stores a permutation of $0,\ldots,n-1$ and for all $1 \leq r < n$, letting $\mathsf{PSA}[r-1] = j$ and $\mathsf{PSA}[r] = k$, we have $x[j\,..\,j+\mathcal{L}[j]-1] \leq x[k\,..\,k+\mathcal{L}[k]-1]$.

Structure of this section. In Section 3.4.2, we provide three $\mathcal{O}(n)$-space algorithms for computing the PSA directly, with time complexities $\mathcal{O}(n\log^2 n)$, $\mathcal{O}(n\log n)$ and $\mathcal{O}(n)$. In Section 3.4.3, we apply our solution to this general problem in the setting of weighted sequences to obtain an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space algorithm for constructing a new $\mathcal{O}(nz)$-sized array index for weighted sequences.

3.4.2 $\mathcal{O}(n)$-space algorithms for computing PSA

Sparse Table-based $\mathcal{O}(n\log^2 n)$-time algorithm

The algorithm presented in this subsection applies a combination of the *Sparse Table* idea for answering RMQs [15] and the *doubling technique* [49] to the context of sorting prefixes of suffixes (factors) of x. Using this combination, one may easily obtain an $\mathcal{O}(n\log n)$-time and $\mathcal{O}(n\log n)$-space algorithm for constructing the PSA [25]. We tweak this solution to require only $\mathcal{O}(n)$ space, suffering an additional multiplicative $\log n$ factor in the time complexity. There are $\mathcal{O}(\log n)$ levels: at the kth level, we sort prefixes of suffixes up to length 2^{k+1}; at each level, $\mathcal{O}(n\log n)$ time is required to sort these factors using any optimal comparison-based sorting algorithm [22].

The aforementioned scheme assumes that we can compare two factors in constant time. To this end, we borrow the Sparse Table algorithm idea for answering RMQs: the minimum value in a given range r is the minimum between the minimums of any two, potentially overlapping, subranges whose union is r. The same idea can be applied in a completely different context:

Fact 3.4.1. *Given two strings x and y, with $|x| \leq |y|$, and $k = \lfloor \log |x| \rfloor$, $x \leq y$ if and only if $(x[0..2^k], x[|x|-2^k..|x|-1]) \leq (y[0..2^k], y[|x|-2^k..|x|-1])$.*

We thus compute the ranks of prefixes of suffixes whose lengths are multiples of two using the doubling technique [49] and then use these ranks to sort prefixes whose lengths may not be multiples of two by applying Fact 3.4.1. Note that this computation can be done level by level in a total of $\mathcal{O}(\log n)$ levels, and therefore the working space is $\mathcal{O}(n)$. We formalise this algorithm, denoted by ST-PSA, in the pseudocode below. We start by initialising the elements in the PSA by sorting and ranking the letters of x (Lines 2–8). We store these ranks in an array (Line 9). Then, at level k (Line 10), we compute the ranks of prefixes whose lengths are multiples of two using the previous level information and radix sort in $\mathcal{O}(n)$ time (Lines 11–12). Next, we sort and rank *all* prefixes up to length 2^{k+1} using a comparison-based sorting algorithm and Fact 3.4.1 in $\mathcal{O}(n \log n)$ time (Lines 13–14). We store these ranks in an array (Line 15) and proceed to the next level. Thus the total time required is $\mathcal{O}(n \log^2 n)$ and the space is $\mathcal{O}(n)$. The value of this algorithm is its practicality: (a) it requires very little space; (b) the number of levels required is in fact $\lfloor \log \ell \rfloor$, where ℓ is the maximum value in $\mathcal{L}$; and (c) at level k it suffices to sort groups of elements having the same rank at level $k-1$.

Algorithm *ST-PSA*$(x, n, \mathcal{L})$

for $i \leftarrow 0$ to $n-1$ **do**

PSA$[i] \leftarrow i$;

Sort PSA using the following comparison rule for PSA$[i]$ and PSA$[j]$:

if $x[i] < x[j]$ **then** PSA$[i] <$ PSA$[j]$;

else if $x[i] > x[j]$ **then** PSA$[i] >$ PSA$[j]$;

else PSA$[i] =$ PSA$[j]$;

Rank the elements of PSA and store their ranks in $\mathsf{Rank}_{\mathsf{PSA}}$;

$\mathsf{Rank}_{\mathsf{PREF}} \leftarrow \mathsf{Rank}_{\mathsf{PSA}}$;

for $k \leftarrow 1$ to $\lfloor \log n \rfloor$ **do**

Construct an array $\mathcal{A}$ of pairs: $\mathcal{A}[i] = (\mathsf{Rank}_{\mathsf{PREF}}[i], \mathsf{Rank}_{\mathsf{PREF}}[i+2^{k-1}])$;

Sort the pairs in $\mathcal{A}$ using radix sort and store their ranks in $\mathsf{Rank}_{\mathsf{CURR}}$;

Sort PSA using $\mathcal{L}$, $\mathsf{Rank}_{\mathsf{PSA}}$, $\mathsf{Rank}_{\mathsf{CURR}}$, and Fact 3.4.1 for the comparison;

Rank the elements of PSA and store their new ranks in $\mathsf{Rank}_{\mathsf{PSA}}$;

$\mathsf{Rank}_{\mathsf{PREF}} \leftarrow \mathsf{Rank}_{\mathsf{CURR}}$;

return PSA;

Average-case $\mathcal{O}(n)$-time algorithm

It should be clear that algorithm ST-PSA attains the time bound of $\mathcal{O}(n \log^2 n)$ for string $x[i] = a$ and $\mathcal{L}[i] = i+1$, for all $0 \leq i < n$. Let us have a more careful analysis of this algorithm in the average-case setting.

Our analysis model We assume that the input is a string x of length n over an alphabet Σ of size $\sigma > 1$ with the letters of x being independent and identically distributed random variables, uniformly distributed over Σ.

Under this model, the expected length of the longest repeated substring of x is known to be $2\log_\sigma n + \mathcal{O}(1)$ [42]. Hence, similar to the average-case analysis for computing the standard suffix array [49], the average-case time complexity of algorithm ST-PSA is $\mathcal{O}(n \log_\sigma n \cdot \log\log_\sigma n)$.

Note that we cannot directly apply the rest of the tricks presented in [49] to shave the $\log\log_\sigma n$ factor. Intuitively, the reason for this extra hardness, compared to the standard setting, is the fact that we need to account for property Π: not all values in $\mathcal{L}$ are at least $2\log_\sigma n + \mathcal{O}(1)$. Notably, we manage here to shave not only the $\log\log_\sigma n$ factor but the $\log_\sigma n$ factor as well. Let us denote this new algorithm by AC-PSA.

The main idea comes from [49]. Let $t = \lfloor \log_\sigma n \rfloor$ and consider mapping each string of length t over Σ to a unique integer obtained when the string is viewed as a t-digit. This is an isomorphism onto the range $[0, \sigma^t - 1] \subseteq [0, n-1]$. We define $s(x[i..i+t-1])$ as the integer signature of $x[i..i+t-1]$. We have that

$$s(x[i..i+t-1]) = s(x[i-1..i+t-2]) \cdot \sigma - x[i-1]\sigma^t + x[i+t-1].$$

It should thus be clear that in time $\mathcal{O}(n)$ we can compute $s(x[i..i+t-1])$ for all i. In our setting, however, we need to account for property Π. We thus go on to generalise this technique as follows.

Let $\Sigma' = \Sigma \cup \{\$\}$, where \$ is a letter (lexicographically) smaller than all letters in Σ. Further let $t' = \lfloor \log_{\sigma+1} n \rfloor$ and consider an analogous mapping onto the range $[0,(\sigma+1)^{t'}-1] \subseteq [0,n-1]$. We define the integer signature s' of $x[i..i+t'-1]]$ as follows:

$$s'(x[i..i+t'-1]]) = \begin{cases} x[i]\cdot(\sigma+1)^{t'-1} + x[i+1]\cdot(\sigma+1)^{t'-2} + \ldots + x[i+t'-1] & : \mathcal{L}[i] \geq t' \\ x[i]\cdot(\sigma+1)^{t'-1} + \ldots + x[i+\mathcal{L}[i]-1](\sigma+1)^{t'-\mathcal{L}[i]} + & \\ \$\cdot(\sigma+1)^{t'-\mathcal{L}[i]-1} + \ldots + \$\cdot(\sigma+1) & : \mathcal{L}[i] < t'. \end{cases} \tag{3.2}$$

The initialisation step consists of computing $s'(x[0..t'-1]])$ trivially in time $\mathcal{O}(t')$.

We first consider the case $\mathcal{L}[i] \geq t'$. We make a first pass, from left to right, and compute $s'(x[i..i+t'-1]])$, by ignoring the properties, as follows.

$$s'(x[i..i+t'-1]]) = (s(x[i-1..i+t'-2]) - x[i-1]\cdot(\sigma+1)^{t'-1})\cdot(\sigma+1) + x[i+t'-1].$$

At this point all signatures are computed correctly for all i such that $\mathcal{L}[i] \geq t'$. It should be clear that this can be implemented in time $\mathcal{O}(n)$.

We then consider the case $\mathcal{L}[i] < t'$. We make a second pass, from left to right. Assuming that \$ is mapped to 0, we mask the $t' - \mathcal{L}[i]$ letters that are not there, due to the property, with 0's; we use the previously computed signatures and standard word-level operations to

achieve that. Let $r = \lceil \log(\sigma+1) \rceil$. We have that:

$$s'(x[i..i+t'-1]]) :=$$

$$(s'(x[i..i+t'-1]])\ \texttt{AND}\ ((2^{r\cdot t'}-1)-(2^{r\cdot(t'-\mathcal{L}[i])}-1)) =$$

$$(s'(x[i..i+t'-1]])\ \texttt{AND}\ (2^{r\cdot t'}-2^{r\cdot(t'-\mathcal{L}[i])}). \quad (3.3)$$

This computation can be executed in $\mathcal{O}(1)$ time since $t' = \lfloor \log_{\sigma+1} n \rfloor$. The whole procedure thus takes time $\mathcal{O}(n)$. This completes the computation of $s'(x[i..i+t-1])$ for all i.

At this point, we apply the doubling technique of [49] on string $x \cdot \$ \ldots \$$ of length $2n$; namely, we append n \$ to x. Instead of performing a radix sort on the first letter of each suffix, we perform it on the t'-length prefixes of suffixes using their signatures. This radix sort requires time $\mathcal{O}(n)$ because $t' = \lfloor \log_{\sigma+1} n \rfloor$. For the second stage, instead of performing a radix sort on the signatures of all $2t'$-length prefixes of suffixes, each suffix is represented by a pair. The first element of this pair is the rank of the t'-length prefix; the second element is the signature of the succeeding substring of length t'. The former (rank) is computed at the previous stage. For the latter (signature), note that, due to the properties, we cannot guarantee that these signatures have been computed at the previous stage. All signatures, however, can be computed in $\mathcal{O}(n)$ time for all suffixes using the aforementioned method. Sorting these pairs can be done in $\mathcal{O}(n)$ time. Since the expected length of the longest repeated substring of x is $t(2+\mathcal{O}(t^{-1}))$, at most three stages of the doubling technique are expected to be required to complete the sort. Thus algorithm AC-PSA solves the PROPERTY SUFFIX ARRAY problem in $\mathcal{O}(n)$ time on average using $\mathcal{O}(n)$ working space.

LCP-based $\mathcal{O}(n \log n)$-time algorithm

The algorithm presented in this subsection is based on the following fact.

Fact 3.4.2. *Given two factors of x, $x[i_1 \,..\, j_1]$ and $x[i_2 \,..\, j_2]$, with $\mathsf{iSA}[i_1] < \mathsf{iSA}[i_2]$, we have that $x[i_2 \,..\, j_2] \leq x[i_1 \,..\, j_1]$ if and only if $j_2 - i_2 \leq \mathsf{lcp}(\mathsf{iSA}[i_1], \mathsf{iSA}[i_2])$ and $j_2 - i_2 \leq j_1 - i_1$.*

Recall that $\mathsf{lcp}()$ queries for two arbitrary suffixes of x can be answered in time $\mathcal{O}(1)$ per query after an $\mathcal{O}(n)$-time preprocessing of the LCP array of x [49, 15]. We can then perform any optimal comparison-based sorting algorithm (use Fact 3.4.2 for the comparison) on the set of prefixes of suffixes. Thus the total time required is $\mathcal{O}(n \log n)$ and the working space is $\mathcal{O}(n)$. We formalise this algorithm, denoted by $\mathsf{LCP\text{-}PSA}$, in the pseudocode below.

Algorithm *LCP-PSA*$(x, n, \mathcal{L})$

Compute $\mathsf{SA}, \mathsf{iSA}, \mathsf{LCP}, \mathsf{RMQ}_{\mathsf{LCP}}$ of x;

for $i \leftarrow 0$ to $n-1$ **do**

$\mathsf{PSA}[i] \leftarrow \mathsf{SA}[i]$;

Sort PSA using the following comparison rule for $\mathsf{PSA}[i]$ and $\mathsf{PSA}[j]$:

if $i < j$ **then** $k \leftarrow \mathsf{RMQ}_{\mathsf{LCP}}(i+1, j)$;

else $k \leftarrow \mathsf{RMQ}_{\mathsf{LCP}}(j+1, i)$;

if $LCP[k] < \min\{\mathcal{L}[SA[i]], \mathcal{L}[SA[j]]\}$ **then**

if $i < j$ **then** $\mathsf{PSA}[i] < \mathsf{PSA}[j]$;

else $\mathsf{PSA}[i] > \mathsf{PSA}[j]$;

else

if $\mathcal{L}[SA[i]] < \mathcal{L}[SA[j]]$ **then** $\mathsf{PSA}[i] < \mathsf{PSA}[j]$;

else $\mathsf{PSA}[i] > \mathsf{PSA}[j]$;

return PSA;

Union-Find-based $\mathcal{O}(n)$-time algorithm

In this section we assume the precomputation of SA, iSA and LCP of x. Given the iSA, the LCP array and $\mathcal{L}$, let $f_i = \max_{0 \leq r \leq \mathsf{iSA}[i]} \{r | \mathsf{LCP}[r] < \mathcal{L}[i]\}$. Informally, f_i tells us how many suffixes are lexicographically smaller than $x[i..i+\mathcal{L}[i]-1]$ (see also Example 3.4.2 in this regard). It follows from the following lemma that in order to construct the PSA it is enough to sort the ordered pairs $(f_i, \mathcal{L}[i])$.

Lemma 3.4.3. *Given two factors of x, $x[i_1 \,..\, j_1]$ and $x[i_2 \,..\, j_2]$, we have that if $(f_{i_1}, j_1 - i_1) \leq (f_{i_2}, j_2 - i_2)$ then $x[i_1 \,..\, j_1] \leq x[i_2 \,..\, j_2]$.*

Proof. Note that $x[i \,..\, j]$ is a prefix of $x[\mathsf{SA}[f_i] \,..\, n-1]$. Thus if

- either $f_{i_1} < f_{i_2}$
- or $f_{i_1} = f_{i_2}$ and $j_1 - i_1 \leq j_2 - i_2$

then we have that $x[i_1 \,..\, j_1] \leq x[i_2 \,..\, j_2]$. □

Example 3.4.2 (Running example)**.** For $i = 3$, we have that $\mathsf{iSA}[3] = 7$, and hence we obtain the pair $(f_3, \mathcal{L}[3]) = (5, 1)$:

i	0	1	2	3	4	5	6	7	8
$\mathcal{L}[i]$	4	3	2	1	3	2	3	2	1
$\mathsf{SA}[i]$	6	7	4	2	0	8	5	3	1
$\mathsf{LCP}[i]$	0	1	2	3	1	0	1	2	0
$\mathcal{L}[\mathsf{SA}[i]]$	3	2	3	2	4	1	2	1	3
$f_{\mathsf{SA}[i]}$	0	1	2	1	4	5	6	5	8
$\mathsf{PSA}[i]$	6	2	7	4	0	3	8	5	1

The computational problem is to compute f_i efficiently for all i; for this we rely on the Union-Find data structure [22] in a similar manner as the authors in [45]. Our technique also resembles the technique by Kociumaka, Radoszewski, Rytter and Waleń for answering off-line weighted ancestor queries in trees. Union-Find maintains a partition of $\{0, 1, \ldots, n-1\}$, where each set has a representative element, and supports three basic operations:

- $\mathsf{MakeSet}(n)$ creates n new sets $\{0\},\{1\},\ldots,\{n-1\}$, where the representative index of set $\{i\}$ is i.
- $\mathsf{Find}(i)$ returns the representative of the set containing i.
- $\mathsf{Union}(i,j)$ first finds the set S_i containing i and the set S_j containing j. If $S_i \neq S_j$, then they are replaced by the set $S_i \cup S_j$.

In the algorithm described below, we only encounter *linear* Union-Find instances, in which the sets of the partition consist of consecutive integers and the representative of each set is its smallest element. We rely on the following result.

Theorem 3.4.4 ([30])**.** *A sequence of q given linear Union and Find operations over a partition of $\{0,1,\ldots,n-1\}$ can be performed in time $\mathcal{O}(n+q)$.*

We perform the following initialisation steps in $\mathcal{O}(n)$ time:

1. Initialise an array $\mathcal{A}$ of linked lists of size n;
2. Initialise the Union-Find data structure by calling $\mathsf{MakeSet}(n)$;
3. Sort indices $\{0,1,\ldots,n-1\}$ based on $\mathcal{L}[i]$ (store them in an array $\mathcal{M}_{\mathcal{L}}$);
4. Sort indices $\{0,1,\ldots,n-1\}$ based on $\mathsf{LCP}[i]$ (store them in an array $\mathcal{M}_{\mathsf{LCP}}$).

Then, for all j from $k = \max\{\max_i\{\mathsf{LCP}[i]\}, \max_i\{\mathcal{L}[i]\}\}$ down to 1 we do the following:

1. $\mathsf{Union}(i-1,i)$ for each i such that $\mathsf{LCP}[i] = j$ using $\mathcal{M}_{\mathsf{LCP}}$;
2. We find all i for which $\mathcal{L}[i] = j$ using $\mathcal{M}_{\mathcal{L}}$ and conclude that $f_i = \mathsf{Find}(\mathsf{iSA}[i])$; we store i at the head of the linked list $\mathcal{A}[f_i]$.

Note that after performing the Union operations for some j, the representative element of the set containing α, $\mathsf{Find}(\alpha)$, is the greatest $\beta \leq \alpha$, for which $\mathsf{LCP}[\beta] \leq j-1$. Thus, in the end of the computation, $\mathcal{A}[j]$ stores the indices i, for which $f_i = j$. In addition, the elements of each list $\mathcal{A}[j]$ are in the order of non-decreasing $\mathcal{L}[i]$. We can thus just read the elements of the linked lists in $\mathcal{A}$ from the left to the right and from the head to the tail to obtain the PSA. We formalise this algorithm, denoted by UF-PSA, in the pseudocode on next page.

Algorithm *UF-PSA*$(x, n, \mathcal{L})$

Compute SA, iSA and LCP of x;

Construct a map $\mathcal{M}_{\mathsf{LCP}}$ such that $\mathcal{M}_{\mathsf{LCP}}[i] = \{j | \mathsf{LCP}[j] = i\}$;

Construct a map $\mathcal{M}_{\mathcal{L}}$ such that $\mathcal{M}_{\mathcal{L}}[i] = \{j | \mathcal{L}[j] = i\}$;

Initialise an array of lists $\mathcal{A}$ of size n;

Initialise a Union-Find data structure $\mathcal{UF}$;

$\mathcal{UF}.\mathsf{MakeSet}(n)$;

$lcp_{\max} \leftarrow \max\{\mathsf{LCP}[0], \mathsf{LCP}[1], \ldots, \mathsf{LCP}[n-1]\}$;

$\ell_{\max} \leftarrow \max\{\mathcal{L}[0], \mathcal{L}[1], \ldots, \mathcal{L}[n-1]\}$;

for $j \leftarrow k = \max\{lcp_{\max}, \ell_{\max}\}$ to 1 **do**

  foreach $i \in \mathcal{M}_{LCP}[j]$ **do**

    $\mathcal{UF}.\mathsf{Union}(i-1, i)$;

  foreach $i \in \mathcal{M}_{\mathcal{L}}[j]$ **do**

    $f \leftarrow \mathcal{UF}.\mathsf{Find}(\mathsf{iSA}[i])$;

    Insert i at the head of $\mathcal{A}[f]$;

for $j \leftarrow 0$ to $n-1$ **do**

  foreach $i \in \mathcal{A}[j]$ **do**

    INSERT(i, PSA);

return PSA;

Example 3.4.3 (Running example)**.** The following two tables show the partition of $\{0,1,\ldots,n-1\}$ before (top) and after (bottom) the Union operations performed for $j=1$. Each monochromatic block represents a set in the partition.

i	0	1	2	3	4	5	6	7	8
$\mathsf{LCP}[i]$	0	1	2	3	1	0	1	2	0

i	0	1	2	3	4	5	6	7	8
$\mathsf{LCP}[i]$	0	1	2	3	1	0	1	2	0
$\mathcal{L}[i]$	4	3	2	1	3	2	3	2	1

Find operations are then performed for those i for which $\mathcal{L}[i]=1$. For example for $i=3$ we have that $\mathsf{Find}(\mathsf{iSA}[3])=\mathsf{Find}(7)=5$, since 5 is the smallest element in the set where 7 belongs. Hence 3 is added in the head of the linked list $\mathcal{A}[5]$.

Putting together Lemma 3.4.3, Theorem 3.4.4 and the above description we obtain the following.

Theorem 3.4.5. *Problem* PROPERTY SUFFIX ARRAY *can be solved in time and space* $\mathcal{O}(n)$.

In the standard setting, the SA is usually coupled with the LCP array to allow for efficient on-line pattern searches (see [49] for the details).

Definition 3.4.6. The *property Longest Common Prefix array* (pLCP) for x and $\mathcal{L}$ is an integer array of size n such that, for all $1\leq r<n$, $\mathsf{pLCP}[r]$ is the length of the longest common prefix of $x[i\,..\,i+\mathcal{L}[i]-1]$ and $x[j\,..\,j+\mathcal{L}[j]-1]$, where $i=\mathsf{PSA}[r]$ and $j=\mathsf{PSA}[r-1]$.

Lemma 3.4.7. *We can compute the pLCP array in time* $\mathcal{O}(n)$.

Proof. We compute the pLCP array while constructing the PSA as follows. If we read both $\mathsf{PSA}[r-1]$ and $\mathsf{PSA}[r]$ from $\mathcal{A}[j]$, we set $\mathsf{pLCP}[r] = \mathcal{L}[\mathsf{PSA}[r-1]]$ since $x[i..i+\mathcal{L}[i]-1]]$ is a prefix of $x[i'..i'+\mathcal{L}[i']-1]]$. Otherwise, we read $\mathsf{PSA}[r-1]$ from $\mathcal{A}[j]$ and $\mathsf{PSA}[r] = i'$ from $\mathcal{A}[j']$ and proceed as follows:

1. If $\mathsf{iSA}[i'] < \mathsf{iSA}[i]$ then $x[i..i+\mathcal{L}[i]-1]$ is a prefix of $x[i'..i'+\mathcal{L}[i']-1]$ and hence we set $\mathsf{pLCP}[r] = \mathcal{L}[i]$;

2. else $\mathsf{iSA}[i] < \mathsf{iSA}[i']$, and since $\mathcal{L}[i] \leq \mathsf{lcp}(j, \mathsf{iSA}[i])$ and $\mathcal{L}[i'] \leq \mathsf{lcp}(j', \mathsf{iSA}[i'])$ we set

$$\mathsf{pLCP}[r] = \min\{\mathsf{lcp}(j, j'), \mathcal{L}[i], \mathcal{L}[i']\}.$$

We can compute $\mathsf{lcp}(j, j')$ for all consecutive non-empty lists $\mathcal{A}[j]$, $\mathcal{A}[j']$ in a simple scan of the LCP array in time $\mathcal{O}(n)$. □

Remark 3.4.8. Alternatively, we can compute the pLCP array using lcp queries, since $\mathsf{pLCP}[r] = \min\{\mathsf{lcp}(\mathsf{PSA}[r-1], \mathsf{PSA}[r]), \mathcal{L}[\mathsf{PSA}[r-1]], \mathcal{L}[\mathsf{PSA}[r]]\}$.

Finally, it is worth noting that the algorithms presented in this section for constructing the PSA depend neither on the fact that $\mathcal{L}[i] \geq \mathcal{L}[i-1]-1$ nor on the fact that we have (at most) one substring starting at each position. As a byproduct we thus obtain the following result *without* the aid of suffix tree, which is interesting in its own right.

Theorem 3.4.9. *Given q substrings of a string x of length n, encoded as intervals over x, we can sort them lexicographically in time $\mathcal{O}(n+q)$.*

3.4.3 Weighted Suffix Array

In this section, we present the algorithm for constructing a new index for a weighted sequence X of length n and a cumulative weighted threshold $1/z$. We combine the ideas presented above with the following powerful combinatorial result (Theorem 3.2.9). Recall that theorem 3.2.9 tells us that one can construct in $\mathcal{O}(nz)$ time a family of $\lfloor z \rfloor$ special weighted sequences, each of length n, that carry all the information about all the strings occurring in X. More specifically, a string occurs with probability $\beta \geq 1/z$ at position i in one of these $\lfloor z \rfloor$ special weighted sequences if and only if it occurs at position i of X with probability β.

Definition 3.4.10. The *Weighted Suffix Array* (WSA) for X and cumulative weighted threshold $1/z$ is an integer array (of size at most $n\lfloor z \rfloor$) storing the path-labels of the terminal nodes of the Weighted Index for X and $1/z$ in the order in which they are visited in a (lexicographic) depth first traversal.

The idea is to create a new special weighted sequence Y by concatenating these $\lfloor z \rfloor$ special weighted sequences. At this point we view Y as the standard string y of length $n\lfloor z \rfloor$ (at most one letter per position has a positive probability). The probabilities at each position of Y and the ends of the original $\lfloor z \rfloor$ special weighted sequences give array $\mathcal{L}$ for Y. We then construct the PSA for y and $\mathcal{L}$.

We are not done yet since a string of length m occurring at a position i of X may occur at several positions $j_0, j_1, \ldots, j_{k-1}$ in y, with $j_p = i \pmod{n}$ and $\mathcal{L}[j_p] = m$ for all $0 \leq p < k$. We naturally want to keep *one* of these occurrences. We do that as follows: we identify maximal intervals $[r,s]$ in the PSA satisfying $\mathcal{L}[\mathsf{PSA}[q]] = \mathsf{pLCP}[t] = m$ for all $r-1 \leq q \leq s$

and $r \leq t \leq s$; for each such interval, we consider all of the indices in $\{\mathsf{PSA}[q] | r-1 \leq q \leq s\}$ modulo n, we bucket sort the residuals, and finally keep one representative for each of them. Doing this for the PSA of y and $\mathcal{L}$ from left to right, we end up with an array of size *at most* $n\lfloor z \rfloor$ that is the WSA for X and $1/z$.

Theorem 3.4.11. *The WSA for a weighted sequence X of length n over an integer alphabet of size σ and a cumulative weighted threshold $1/z$ can be constructed in $\mathcal{O}(nz)$ time.*

The WSA for X and $1/z$, coupled with the naturally defined *weighted Longest Common Prefix array* (wLCP), which can be inferred directly from the pLCP array of y and $\mathcal{L}$, is an index with comparable capabilities as the ones of the SA coupled with the LCP array in the standard setting [49].

Example 3.4.4. Let $X = [(\mathtt{a},0.5),(\mathtt{b},0.5)]\mathtt{bab}[(\mathtt{a},0.5),(\mathtt{b},0.5)][(\mathtt{a},0.5),(\mathtt{b},0.5)]$ and $\frac{1}{z} = 1/4$. The family of z strings and the corresponding index are as follows:

i	0	1	2	3	4	5
	a	b	a	b	b	b
	a	b	a	b	a	b
	b	b	a	b	b	a
	b	b	a	b	a	a

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
$y[i]$	a	b	a	b	b	b	a	b	a	b	a	b	b	b	a	b	b	a	b	b	a	b	a	a
WSA$[i]$	17	22	10	20	8	6	0	14	2	5	16	21	9	19	7	13	1	4	15	18	12	3		
$\mathcal{L}$[WSA$[i]$]	1	2	2	4	4	5	5	4	4	1	2	3	3	5	5	5	5	2	3	5	5	3		
wLCP$[i]$	0	1	1	2	3	4	4	2	3	0	1	2	2	3	4	3	4	1	2	3	4	2		

3.5 Remark and Experimental Results

In this chapter, we present one algorithm to construct weighted suffix tree and four algorithms for weighted suffix array. Given a weighted string of length n and a cumulative weight threshold $1/z$, the time and space complexity are shown in the following table.

Algorithm	Time	Space
PST	$\mathcal{O}(nz)$	$\mathcal{O}(nz)$
ST-PSA	$\mathcal{O}(nz\log^2 nz)$	$\mathcal{O}(nz)$
LCP-PSA	$\mathcal{O}(nz\log nz)$	$\mathcal{O}(nz)$
UF-PSA	$\mathcal{O}(nz)$	$\mathcal{O}(nz)$
AC-PSA	$\mathcal{O}(nz)$	$\mathcal{O}(nz)$

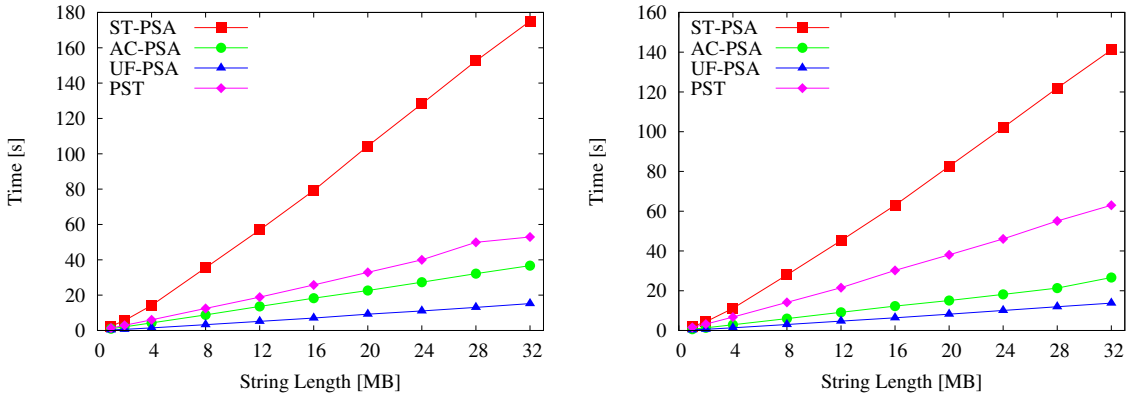
We have implemented algorithms PST to construct the property suffix tree and weighted suffix tree, and ST-PSA, AC-PSA, and UF-PSA to compute the property suffix array and weighted suffix array. The programs have been implemented in the C++ programming language and developed under the GNU/Linux operating system. The input parameters for all programs are a weighted sequence of length n and a positive number z for the cumulative weighted threshold $1/z$. The output of PST is the weighted suffix tree and the output of ST-PSA, AC-PSA and UF-PSA is the weighted suffix array. The PST source code is distributed at https://bitbucket.org/kociumaka/weighted_index and the three PSA source code is distributed at https://github.com/YagaoLiu/WSA under the GNU General Public License. All experiments have been conducted on a Desktop PC using one core of Intel Xeon CPU E5-2640 at 2.60GHz. All programs have been compiled with g++ version 6.2.0 at optimization level 3 (-O3).

It is well-known, among practitioners and elsewhere, that optimal RMQ data structures for on-line $\mathcal{O}(1)$-time querying carry high constants in their preprocessing and querying time [6]. One would not thus expect that algorithm LCP-PSA performs well in practice. Indeed, we have implemented LCP-PSA but we omit its presentation here since it was too slow for the same inputs.

To evaluate the time and space performance of our implementations, we used synthetic and real weighted sequences over the DNA alphabet ($\sigma = 4$).

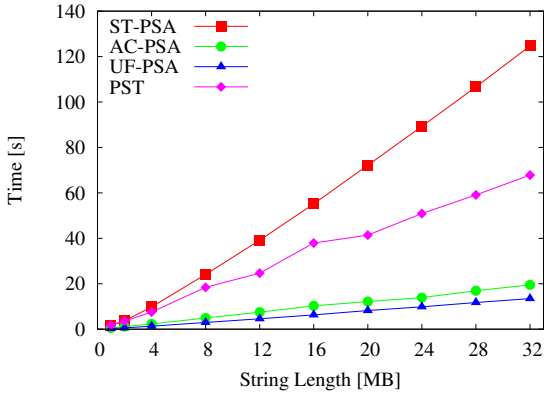
Synthetic weighted DNA sequences In the first experiment, we used synthetic weighted DNA sequences. The weighted sequences were of length ranging from $125,000$ to $4,000,000$. For each length, four different degeneracy percentages, denoted by δ, were used: 1%, 5%, 10% and 20% (percentage of positions where at least two letters with positive probability exist). The probability threshold was set to $1/8$. The strings obtained from the weighted sequences were thus of length ranging from $1,000,000$ to $32,000,000$. The results are plotted in Figures 3.3 and 3.4. In Figure 3.3 we observe that: (i) AC-PSA, UF-PSA and PST run in *linear* time; (ii) ST-PSA runs in (slightly) *super-linear* time; (iii) all four implementations run faster when δ increases. Observations (i) and (ii) confirm the theoretical findings. Observation (iii) is explained by the fact that when δ increases, the Π-valid intervals over the string are shorter on average, and thus fewer comparisons are generally required to resolve ties and thus to obtain the final order. In Figure 3.4 we observe that: (i) all four implementations run in *linear* space; (ii) PST is by far the most space *inefficient* of the four implementations; (iii) ST-PSA is the most space *efficient* of the three implementations. Observation (i) confirms

the theoretical findings. Observations (ii) and (iii) are easily explained by the hidden constant factors in the corresponding space complexities.

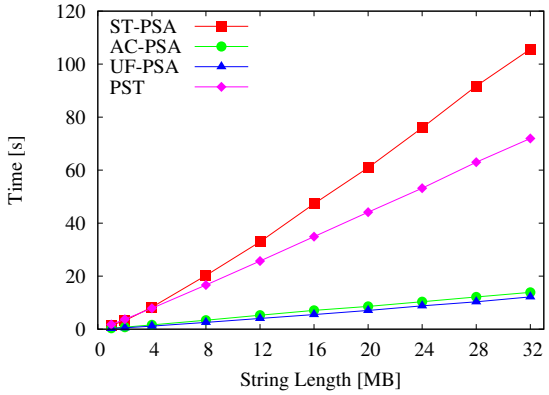


(a) $\delta = 1\%$

(b) $\delta = 5\%$

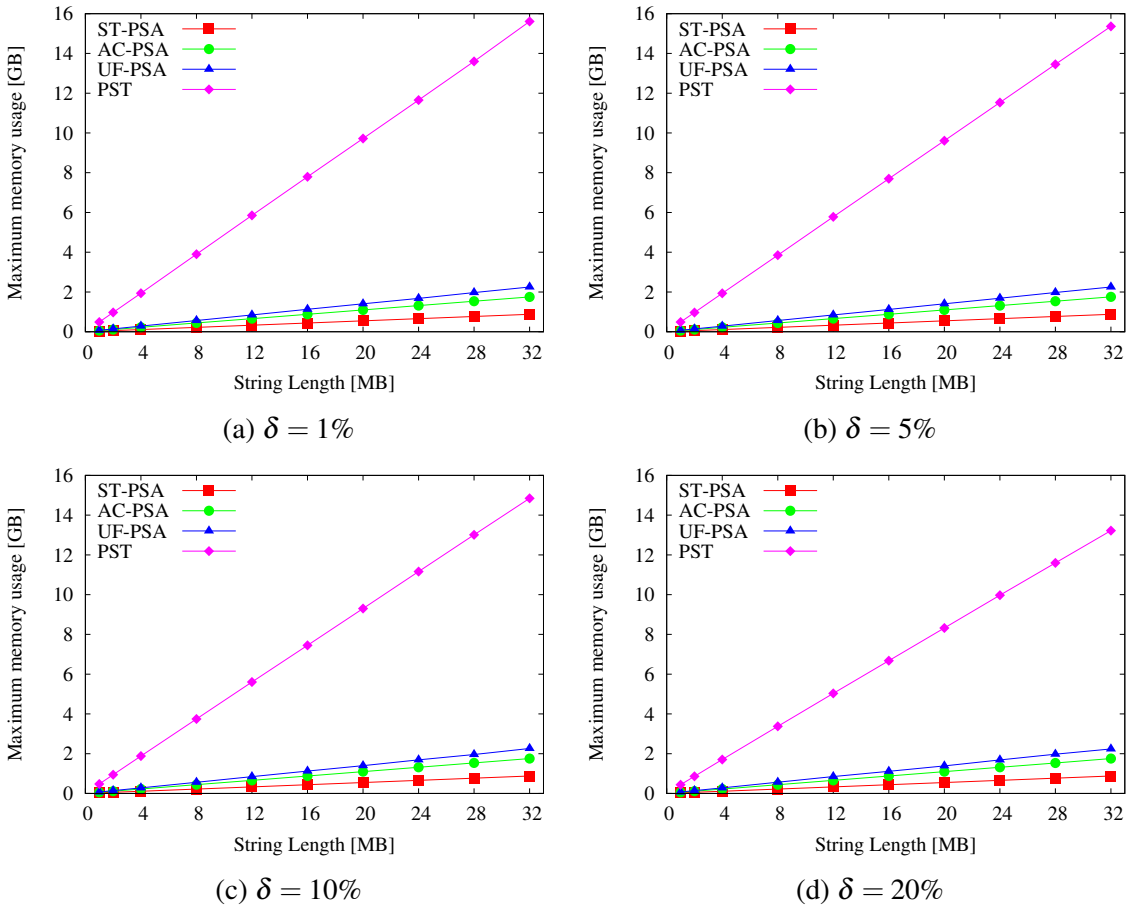


(c) $\delta = 10\%$



(d) $\delta = 20\%$

Fig. 3.3 Elapsed time of ST-PSA, AC-PSA, UF-PSA, and PST using synthetic texts of length ranging from 1MB to 32MB over the DNA alphabet.



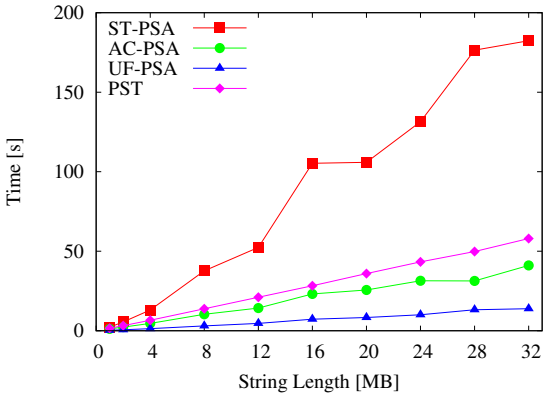
(a) $\delta = 1\%$ (b) $\delta = 5\%$

(c) $\delta = 10\%$ (d) $\delta = 20\%$

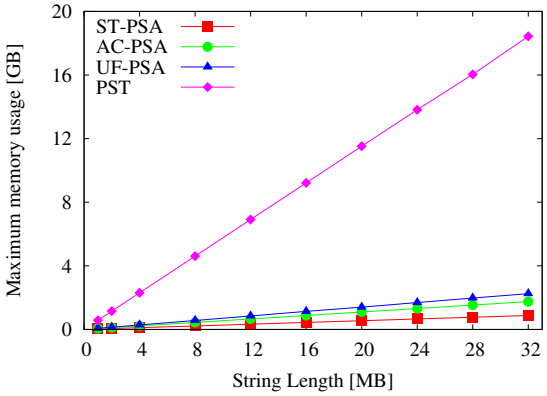
Fig. 3.4 Peak memory usage of ST-PSA, AC-PSA, UF-PSA, and PST using synthetic texts of length ranging from 1MB to 32MB over the DNA alphabet.

Real weighted DNA sequences In the second experiment, we created real weighted DNA sequences by combining the Genome Reference Consortium Human Build 37 (GRCh37) with the variants obtained from the 1000 Genomes Project (October 2011 Integrated Variant Set release) [1]. Specifically we made use of human chromosome 21 data. We randomly extracted fragments of length ranging from $125,000$ to $4,000,000$ from the generated weighted sequence. The probability threshold was set to $1/8$. The results are plotted in Figure 3.5. In both plots (time and space) we observe that the performance is analogous to the performance

with the synthetic data of $\delta = 1\%$. This is explained by the fact that δ is found to be 0.7% in the weighted sequence of chromosome 21.



(a) Elapsed time



(b) Peak memory usage

Fig. 3.5 Elapsed time and peak memory usage of ST-PSA, AC-PSA, UF-PSA, and PST using random real DNA sequences of length ranging from 1MB to 32MB.

Whole-chromosome weighted DNA sequences In the third experiment, we applied our three algorithms, ST-PSA, AC-PSA and UF-PSA, on whole-chromosome weighted DNA sequences. Specifically we combined human (GRCh37) chromosomes 18 to 22 and the corresponding variants (October 2011 Integrated Variant Set release). The weighted sequences were constructed in the same way as in the second experiment. In addition, long prefixes (or suffixes) of the chromosome sequences consisting solely of unknown bases (letter `N`) were

Human Chromosome	Property String Length(MB)	ST-PSA	AC-PSA	UF-PSA
18	624	11512	4086	1446
19	472	7434	1704	1207
20	503	5013	714	422
21	309	2816	486	260
22	281	2706	371	160

(a) Elapsed time(second) of ST-PSA, AC-PSA and UF-PSA on human chromosome 18-22

Human Chromosome	Property String Length(MB)	ST-PSA	AC-PSA	UF-PSA
18	624	17.066	34.132	60.533
19	472	12.921	25.842	50.127
20	503	13.763	27.524	36.052
21	309	8.469	16.937	22.281
22	281	7.701	15.401	19.778

(b) Peak memory usage(GB) of ST-PSA, AC-PSA and UF-PSA on human chromosome 18-22

Table 3.2 Elapsed time and peak memory usage of ST-PSA, AC-PSA and UF-PSA on human chromosome 18-22.

omitted. The performance results are depicted in Table 3.2. As shown before, (i) UF-PSA is the fastest but the most space inefficient; (ii) ST-PSA is the most space efficient but the slowest; and (iii) AC-PSA lies in between as a reasonable trade off. We stress that it was not possible to apply PST due to its memory requirements which far exceed the capacity (64GB) of the machine used.

All the presented experimental results above confirm *fully* our theoretical findings and justify the motivation for the contributions in this thesis.

Chapter 4

Applications of Weighted Index

In this chapter we present a solution for the problem smallest maximal palindromic factorisation using our weighted index.

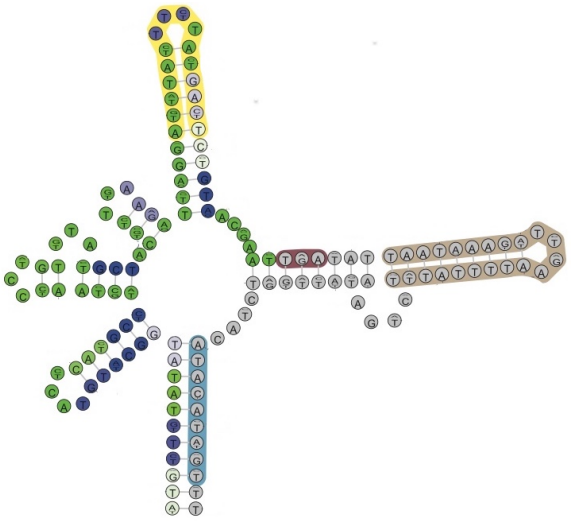
4.1 Introduction

A palindrome is a sequence that reads the same from left to right as from right to left. Detection of palindromic factors in texts is a classical and well-studied problem in algorithms on strings and combinatorics on words with a lot of variants arising out of different practical scenarios. In molecular biology, for instance, palindromic sequences are extensively studied: they are often distributed around promoters, introns, and untranslated regions, playing important roles in gene regulation and other cell processes (see e.g. [5]). In particular these are strings of the form $s\bar{s}^R$, also known as complemented palindromes, occurring in single-stranded DNA or, more commonly, in RNA, where s is a string and $\bar{s}^R$ is the reverse

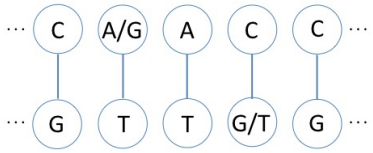
complement of s. In DNA, C-G are complements and A-T are complements; in RNA, C-G are complements and A-U are complements.

A string $x = x[0]x[1]\ldots x[n-1]$ is said to have an initial palindrome of length k if its prefix of length k is a palindrome. Manacher first discovered an on-line algorithm that finds all initial palindromes in a string [48]. Later Apostolico et al observed that the algorithm given by Manacher is able to find all maximal palindromic factors in the string in $\mathcal{O}(n)$ time [10]. Gusfield gave an off-line linear-time algorithm to find all maximal palindromes in a string and also discussed the relation between biological sequences and gaped palindromes (i.e. strings of the form $sv\bar{s}^R$ where the complemented palindromes are separated by v) [32].

The problem that gained significant attention recently is the factorisation of a string x of length n into a sequence of palindromes. We say that $x_1, x_2, \ldots, x_\ell$ is a (maximal) palindromic factorisation of string x, if every x_i is a (maximal) palindrome, $x = x_1 x_2 \ldots x_\ell$, and ℓ is minimal. In biological applications we need to factorise a sequence into palindromes in order to identify *hairpins*, patterns that occur in single-stranded DNA or, more commonly, in RNA. Alatabbi et al gave an off-line $\mathcal{O}(n)$-time algorithm for finding a maximal palindromic factorisation of x [4]. Fici et al presented an on-line $\mathcal{O}(n \log n)$-time algorithm for computing a palindromic factorisation of x [28]; a similar algorithm was presented by I et al [36]. In addition, Rubinchik and Shur [58] devised an $\mathcal{O}(n)$-sized data structure that helps locating palindromes in x; they also showed how it can be used to compute a palindromic factorisation of x in $\mathcal{O}(n \log n)$ time.



(a) Hairpins common to *Malvastrum yellow vein* virus, *Cotton leaf curl Multan* virus isolate, and *Bhendi yellow vein India* virus; figure taken from [51].



(b) Hairpin represented as a weighted string: `C[(A,0.5),(G,0.5)]ACC` (top) and `GTT[(G,0.5),(T,0.5)]G` (bottom).

Fig. 4.1 Hairpins that are common to a set of closely-related sequences can be represented compactly as weighted strings.

Our Problem. In this section, we consider the palindrome problem on weighted sequence. Muhire et al [51] showed how a set of virus species can be clustered using multiple sequence alignment (MSA) to obtain subsets of viruses that have common hairpin structure (see Fig. 4.1(a)). A more compact representation of an MSA can be trivially obtained using

weighted sequence (see Fig. 4.1(b)). The non-trivial computational problem thus arising is how to factorise a weighted string in a sequence of palindromes.

Our Contribution. In this section, we generalise Alatabbi et al's solution for standard strings [4] to compute a maximal palindromic factorisation of a weighted string. In particular, we provide an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space off-line algorithm, where n is the length of the weighted string and $1/z$ is the given threshold. Along the way, we provide an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space off-line algorithm for computing maximal palindromes in weighted strings.

4.2 Preliminaries

Let $x = x[0]x[1]\ldots x[n-1]$ be a *string* of length $|x| = n$ over a finite ordered alphabet Σ of size $|\Sigma| = \sigma = \mathcal{O}(1)$. We denote the *reversal* of x by string x^R, i.e. $x^R = x[n-1]x[n-2]\ldots x[0]$.

A string w is said to be a *palindrome* if and only if $w = w^R$. If factor $x[i\,..\,j]$, $0 \leq i \leq j \leq n-1$, of string $x[0\,..\,n-1]$ is a palindrome, then $\frac{i+j}{2}$ is the *centre* of $x[i\,..\,j]$ in x and $\frac{j-i+1}{2}$ is the *radius* of $x[i\,..\,j]$. Moreover, $x[i\,..\,j]$ is called a *palindromic factor*. It is said to be a *maximal palindrome* if there is no other palindrome in x with centre $\frac{i+j}{2}$ and larger radius. Hence x has exactly $2n-1$ maximal palindromes. A maximal palindrome w can be encoded as a pair (c,r), where c is the centre of w and r is the radius of w. By $\mathcal{MP}(x)$, we denote the set of centre-distinct maximal palindromes of string x. The sequence $x_1, x_2, \ldots, x_\ell$ of ℓ non-empty strings is a *(maximal) palindromic factorisation* of a string x if all strings x_i are (maximal) palindromes, $x = x_1x_2\ldots x_\ell$, and ℓ is minimal.

Definition 4.2.1. Given a cumulative weight threshold $1/z \in (0,1]$, a weighted string X of length m is a *z-palindrome* if and only if there exists at least one z-valid factor u of X of length m which is a palindrome.

Example 4.2.1. Let $X = \texttt{a[(a,0.5),(b,0.5)]bab[(a,0.4),(b,0.6)]a}$ of length $m = 7$ and $1/z = 1/8$. $u = \texttt{abbabba}$ is a z-valid factor of X of length 7 and u is a palindrome. Hence we say X is a z-palindrome.

If the weighted string $X[i\,..\,j]$ is a z-palindrome, we analogously define the number $\frac{i+j}{2}$ as the centre of $X[i\,..\,j]$ in X and $\frac{j-i+1}{2}$ as the radius of $X[i\,..\,j]$.

Definition 4.2.2. Let X be a weighted string of length n, $1/z \in (0,1]$ a cumulative weight threshold, and $X[i\,..\,j]$, where $0 \leq i \leq j \leq n-1$, a z-palindrome. Then $X[i\,..\,j]$ is a *maximal z-palindrome* if there is no other z-palindrome in X with centre $\frac{i+j}{2}$ and larger radius.

A maximal z-palindrome can thus also be encoded as a pair (c,r). We study the following computational problem.

SMALLEST MAXIMAL z-PALINDROMIC FACTORISATION
Input: A weighted string X of length n and a cumulative weight threshold $1/z \in (0,1]$
Output: $X_1, X_2, \ldots, X_\ell$, if any, such that $X = X_1X_2\ldots X_\ell$ and X_i, for all $1 \leq i \leq \ell$, is a maximal z-palindrome, and ℓ is minimal.

We call this output sequence $X_1, X_2, \ldots, X_\ell$, i.e. when ℓ is minimal, a *smallest maximal z-palindromic factorisation* of X.

Recall that the suffix tree $\mathcal{T}(x)$ of a non-empty string x of length n is a compact trie representing all suffixes of x. The suffix tree of a string of length n can be computed in time and space $\mathcal{O}(n)$ [27]. It can also be preprocessed in time and space $\mathcal{O}(n)$ so that *lowest*

common ancestor (LCA) queries for any pair of explicit nodes can be answered in $\mathcal{O}(1)$ time per query [15].

Fact 4.2.3 ([32])**.** *Given a string x, $\mathcal{MP}(x)$ can be computed in time $\mathcal{O}(|x|)$.*

4.3 $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space algorithm

In this section, we present an algorithm to compute a smallest maximal z-palindromic factorisation of a given weighted string X of length n for a given cumulative threshold $1/z \in (0,1]$. Our algorithm follows the one of Alatabbi et al for computing a smallest maximal palindromic factorisation of standard strings [4] with some crucial modifications.

Why the algorithm of Alatabbi et al cannot be applied for weighted strings. Odd-length maximal palindromes centered at position i of a standard string x can be computed by finding the longest common prefix of suffixes $x[i\,..\,n-1]$ and $x^R[n-i-1\,..\,n-1]$. The longest common prefix of two suffixes can be found in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$-time pre-processing of the suffix tree of $x\#x^R\$$, where $\#, \$ \notin \Sigma$, using LCA queries; using a similar computation, we can find all even-length maximal palindromes (see [32] for the details).

The length of the longest common z-valid prefix of any two suffixes of our weighted string X can be computed in time $\mathcal{O}(z)$ after $\mathcal{O}(nz)$-time pre-processing using weighted suffix tree (inspect also Figure 4.2). However, this does not guarantee that the two corresponding common z-valid prefixes shall form a maximal z-palindrome: the two prefixes are z-valid by definition of the weighted suffix tree but their concatenation that forms a palindrome *may not* be z-valid because its occurrence probability drops below $1/z$.

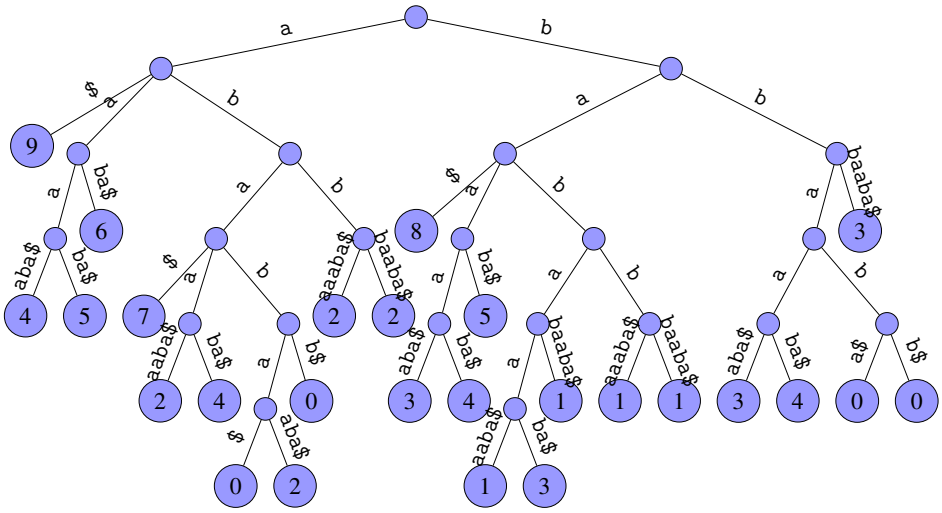


Fig. 4.2 The WST for X and $1/z$ shown in Example 4.3.1 (labels of edges to terminal nodes are appended with a letter $\$ \notin \Sigma$ for convenience).

We hence proceed as follows. By $\mathcal{MP}(X,z)$, we denote the set of centre-distinct maximal z-palindromes of our weighted string X. Recall that we can represent a z-palindrome with centre c and radius r by (c,r).

We construct the z-estimation of X, denoted by $\mathcal{Z}_X$, with property Π. Since $\mathcal{Z}_X$ is a concatenation of $\lfloor z \rfloor$ estimation substrings of length n, by $\mathcal{Z}_k$, $0 \leq k < \lfloor z \rfloor$. we denote each estimation substring.

Example 4.3.1. Given the weighted string

$$X = \texttt{[(a,0.5),(b,0.5)]bab[(a,0.5),(b,0.5)][(a,0.75),(b,0.25)]aaba}$$

and a cumulative weight threshold $1/z = 1/4$, we have:

i	0	1	2	3	4	5	6	7	8	9
$\mathcal{Z}_0[i]$	a	b	a	b	a	a	a	a	b	a
$\Pi_0[i]$	5	9	8	7	6	5	4	3	2	1
$\mathcal{Z}_1[i]$	a	b	a	b	b	a	a	a	b	a
$\Pi_1[i]$	5	4	3	2	1	5	4	3	2	1
$\mathcal{Z}_2[i]$	b	b	a	b	a	a	a	a	b	a
$\Pi_2[i]$	5	9	8	7	6	5	4	3	2	1
$\mathcal{Z}_3[i]$	b	b	a	b	b	a	a	a	b	a
$\Pi_3[i]$	5	4	3	2	1	5	4	3	2	1

Lemma 4.3.1 (Section 3.2)**.** *Given a weighted string X of length n and a cumulative weight threshold $1/z \in (0,1]$, the z-estimation $\mathcal{Z}_X$ and property Π of X can be constructed in time and space $\mathcal{O}(nz)$.*

Fact 4.3.2. *Given a weighted string X of length n and a cumulative weight threshold $1/z \in (0,1]$, we have that $\mathcal{MP}(X,z) \subseteq \mathcal{MP}(\mathcal{Z}_0,z) \cup \mathcal{MP}(\mathcal{Z}_1,z) \cup \ldots \cup \mathcal{MP}(\mathcal{Z}_{\lfloor z \rfloor -1},z)$.*

Proof. Suppose $U = X[i..j]$ is a maximal z-palindrome of centre $c = \frac{i+j}{2}$ and radius $r = \frac{j-i+1}{2}$. By definition of U there must exist a z-valid palindromic factor u of U of radius r. Therefore, by definition of the special-weighted strings of X, u must be a z-valid factor of some $\mathcal{Z}_k$ and thus $(c,r) \in \mathcal{MP}(\mathcal{Z}_k,z)$. □

There are two steps for the correct computation of $\mathcal{MP}(X,z)$. First, we compute the set $\mathcal{A}_k$ of all maximal palindromes of the heavy string of $\mathcal{Z}_k$, for all $0 \leq k < \lfloor z \rfloor$, using Fact 4.2.3. We then need to adjust the radius of each reported palindrome for $\mathcal{Z}_k$ to ensure that it is

z-valid in X (the centre should not change). To achieve this, we compute an array $\mathcal{R}_k$, for each $\mathcal{Z}_k$, such that $\mathcal{R}_k[2c]$ stores the radius of the longest factor at centre c in $\mathcal{Z}_k$ which is a z-valid factor of X at centre c, e.g. $\mathcal{R}_k[2c] = \frac{j-i+1}{2}$, $c = (i+j)/2$, if $\mathcal{Z}_k[i..j]$ is a z-valid factor of X centred at c, and $\mathcal{Z}_k[i-1..j+1]$ is not a z-valid factor of X. By Fact 4.3.2, we cannot guarantee that all (c,r) in $\mathcal{MP}(\mathcal{Z}_k,z)$ are necessarily in $\mathcal{MP}(X,z)$. Hence, the second step is to compute $\mathcal{MP}(X,z)$ from $\mathcal{MP}(\mathcal{Z}_k,z)$ by taking the maximum radius per centre and filtering out everything else.

Lemma 4.3.3. *Given a weighted string X of length n, a cumulative weight threshold $1/z \in (0,1]$, and the special-weighted strings $\mathcal{Z}_X$ of X, each $\mathcal{R}_k$, $0 \le k < \lfloor z \rfloor$, can be computed in time $\mathcal{O}(n)$.*

Proof. By $< i,c,j >$, where $0 \le i \le c \le j \le n-1$, we denote a factor of $\mathcal{Z}_k$ that has starting position i, ending position j and centre $c = (i+j)/2$. We further denote the length of $< i,c,j >$ in $\mathcal{Z}_k$ by $\mathcal{L}_{<i,c,j>}$. A factor $< i,c,j >$ of $\mathcal{Z}_k$ is called a special maximal z-valid factor of $\mathcal{Z}_k$ if $\mathcal{L}_{<i,c,j>} \le \Pi_k[i]$ and $\mathcal{L}_{<i-1,c,j+1>} > \Pi_k[i-1]$.

For each $\mathcal{Z}_k$, we compute $\mathcal{R}_k$ from left to right. If we have $\Pi_k \ge 1$, we set $\mathcal{R}_k[0] = \frac{1}{2}$. If not, we go to the next position until we find a valid letter, say at position ℓ; then we have $\mathcal{R}_k[0] = \cdots = \mathcal{R}_k[2\ell-1] = 0$ and $\mathcal{R}_k[2\ell] = \frac{1}{2}$. Note that this corresponds to the first special maximal z-valid factor. Suppose we have a special maximal z-valid factor $< i,c,j >$ and $\mathcal{R}_k[2c] = \frac{j-i+1}{2}$, we show how to compute $\mathcal{R}_k[2c+1]$, which is the length of the special maximal z-valid factor at centre $c' = \frac{2c+1}{2}$. We add the letter after $< i,c,j >$, so we have $< i,c',j+1 >$ and $\mathcal{L}_{<i,c',j+1>} = \mathcal{L}_{<i,c,j>+1}$ If $\mathcal{L}_{<i,c',j+1>} \le \Pi_k[i]$, the special maximal z-valid factor at centre c' should be $< i,c',j+1 >$ and $\mathcal{R}_k[2c+1] = \mathcal{R}_k[2c] + \frac{1}{2} = \frac{j-i+2}{2}$.

Factor $< i-1, c', j+2 >$ cannot be z-valid, since if $\mathcal{L}_{<i-1,c',j+2>} \leq \Pi_k[i-1]$, we must have $\mathcal{L}_{<i-1,c,j+1>} < \mathcal{L}_{<i-1,c',j+2>} \leq \Pi_k[i-1]$, which gives a longest special maximal z-valid at centre c, namely $< i-1, c, j+1 >$, a contradiction. For $\mathcal{L}_{<i,c',j+1>} > \Pi_k[i]$, the special maximal z-valid factor at centre c' is $< i+1, c', j >$ since it always holds that $\mathcal{L}_{<i+1,c',j>} = \mathcal{L}_{<i,c,j>} - 1 \leq \Pi_k[i] - 1 \leq \Pi_k[i+1]$. Therefore $\mathcal{R}_k[2c+1] = \mathcal{R}_k[2c] - \frac{1}{2} = \frac{j-i}{2}$.

Each centre needs only to be considered once and there exist $2n-1$ distinct centres in each $\mathcal{Z}_k$. Therefore each $\mathcal{R}_k$ can be computed in $\mathcal{O}(n)$ time. □

Fact 4.3.4 (Trivial)**.** *Let $x[i..j]$ be a palindrome of string x with centre c and let u, $|u| < j-i+1$, be a factor of x with centre c. Then u is also a palindrome.*

After computing $\mathcal{A}_k$ and $\mathcal{R}_k$, we perform the following check for each palindrome $(c,r) \in \mathcal{A}_k$. If $r > \mathcal{R}_k[2c]$, the palindrome with radius r is not z-valid but the factor with radius $\mathcal{R}_k[2c]$ is z-valid and maximal (by definition) and palindromic (by Fact 4.3.4); if $r \leq \mathcal{R}_k[2c]$, the palindrome with radius r_i must be z-valid and it is maximal. Therefore we set $(c,r) \in \mathcal{MP}(\mathcal{Z}_k, z)$, such that $r = \min\{r, \mathcal{R}_k[2c]\},\ 0 \leq 2c \leq 2n-2,$ and $r \geq 1/2$.

To go from $\mathcal{MP}(\mathcal{Z}_k, z)$ to $\mathcal{MP}(X, z)$ we need to take the maximum radius for each centre. Therefore for each centre $c/2$, $0 \leq c \leq 2n-2$, we set $(c/2, r) \in \mathcal{MP}(X, z)$, such that $r = \max\{r_k | (c/2, r_k) \in \mathcal{MP}(\mathcal{Z}_k, z), 0 \leq k < \lfloor z \rfloor\}$. We thus arrive at the first result of this article.

Theorem 4.3.5. *Given a weighted string X of length n and a cumulative weight threshold $1/z \in (0,1]$, all maximal z-palindromes in X can be computed in time and space $\mathcal{O}(nz)$.*

After the computation of $\mathcal{MP}(X,z)$, we are in a position to apply the algorithm by Alatabbi et al [4] to find a smallest maximal z-palindromic factorisation. We define a list $\mathcal{F}$ such that $\mathcal{F}[i]$, $0 \leq i \leq n-1$, stores the set of the lengths of all maximal z-palindromes ending at position i in X. We also define a list $\mathcal{U}$ such that $\mathcal{U}[i]$, $0 \leq i \leq n-1$, stores the set of positions j, such that $j+1$ is the starting position of a maximal z-palindrome in X and i is the ending position of this z-palindrome. Thus for a given $\mathcal{F}[i] = \{\ell_0, \ell_1, \ldots, \ell_q\}$, we have that $\mathcal{U}[i] = \{i-\ell_0, i-\ell_1, \ldots, i-\ell_q\}$. Note that $\mathcal{U}[i]$ can contain a "-1" element if there exists a maximal z-palindrome starting at position 0 and ending at position i. Note that the number of elements in $\mathcal{MP}(X,z)$ is at most $2n-1$, and, hence, $\mathcal{F}$ and $\mathcal{U}$ can contain at most $2n-1$ elements. The lists $\mathcal{F}$ and $\mathcal{U}$ can be computed trivially from $\mathcal{MP}(X,z)$.

Finally, we define a directed graph $\mathcal{G}_X = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{i \mid -1 \leq i \leq n-1\}$ and $\mathcal{E} = \{(i,j) \mid j \in \mathcal{U}[i]\}$. Note that (i,j) is a directed edge from i to j. We do a breath first search on $\mathcal{G}_X$ assuming the vertex $n-1$ as the source and identify the shortest path from $n-1$ to -1, which gives a factorisation.

Algorithm *SMPF*$(X, n, 1/z)$

Construct the z-estimation strings $\mathcal{Z}_X$ with property Π of X;

foreach $\mathcal{Z}_k \in \mathcal{Z}_X$ **do**

$\mathcal{A}_k \leftarrow$ maximal palindromes of each $\mathcal{Z}_k$ in $\mathcal{Z}_X$;

Compute $\mathcal{R}_k$ for $\mathcal{Z}_k$;

$\mathcal{MP}(\mathcal{Z}_k, z) \leftarrow$ EMPTYLIST();

foreach $(c, r) \in \mathcal{A}_k$ **do**

$r \leftarrow \min\{r, \mathcal{R}_k[2c]\}$;

if $r \geq \frac{1}{2}$ **then** Insert (c, r) in $\mathcal{MP}(\mathcal{Z}_k, z)$;

$\mathcal{MP}(X, z) \leftarrow$ EMPTYLIST();

foreach $c \in [0, 2n-2]$ **do**

$r \leftarrow \max\{r_k | (c/2, r_k) \in \mathcal{MP}(\mathcal{Z}_k, z), 0 \leq k < \lfloor z \rfloor\}$;

Insert $(c/2, r)$ in $\mathcal{MP}(X, z)$;

$\mathcal{F} \leftarrow$ EMPTYLIST();

$\mathcal{U} \leftarrow$ EMPTYLIST();

foreach $(c, r) \in \mathcal{MP}(X, z)$ **do**

$j \leftarrow \lfloor c + r \rfloor$;

Insert $2r$ in $\mathcal{F}[j]$;

Insert $j - 2r$ in $\mathcal{U}[j]$;

Construct directed graph $\mathcal{G}_X = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{i \mid -1 \leq i \leq n-1\}$,

$\mathcal{E} = \{(i, j) \mid j \in \mathcal{U}[i]\}$ and (i, j) is a directed edge from i to j;

Breadth first search on $\mathcal{G}_X$ assuming the vertex $n-1$ as the source;

Identify the shortest path $P \equiv \langle n-1 = p_\ell, p_{\ell-1}, \ldots, p_2, p_1, p_0 = -1 \rangle$;

Return $X[0..p_1], X[p_1+1..p_2], \ldots, X[p_{\ell-1}+1..p_\ell]$;

We formally present the above as Algorithm SMPF for computing a smallest maximal z-palindromic factorisation and obtain the following result.

Theorem 4.3.6. *Given a weighted string X of length n and a cumulative weight threshold $1/z \in (0,1]$, Algorithm SMPF correctly solves the problem* SMALLEST MAXIMAL z-PALINDROMIC FACTORISATION *in time and space $\mathcal{O}(nz)$.*

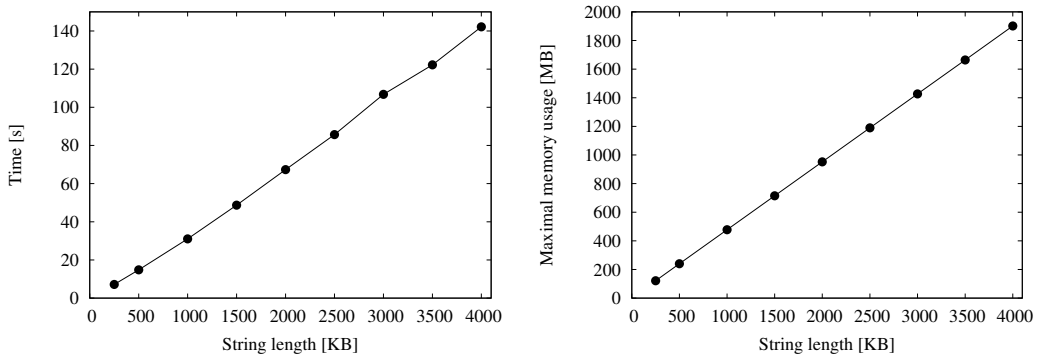
Proof. The correctness follows from Theorem 4.3.5 for computing $\mathcal{MP}(X,z)$ and from the correctness of the algorithm in [4] for computing a smallest maximal palindromic factorisation.

By Lemma 4.3.1, the construction of the special-weighted strings can be done in time and space $\mathcal{O}(nz)$. Computing $\mathcal{A}_k$ and $\mathcal{R}_k$, for all $0 \leq k < \lfloor z \rfloor$, can be done in total time $\mathcal{O}(nz)$ by Fact 4.2.3 and Lemma 4.3.3, respectively. From there on, computing $\mathcal{MP}(X,z)$ can be done in time $\mathcal{O}(nz)$. The lists $\mathcal{F}$ and $\mathcal{U}$ can be computed in time $\mathcal{O}(n)$ since the size of $\mathcal{MP}(X,z)$ is no more than $2n-1$. There exist in total $n+1$ vertices in $\mathcal{G}_X$. The number of edges $|\mathcal{E}|$ depends on $\mathcal{U}$, which contains no more than $2n$ elements; we have $|\mathcal{E}| = \mathcal{O}(n)$. Therefore, the construction of $\mathcal{G}_X$ and the breadth first search can be done in time $\mathcal{O}(|\mathcal{V}|+|\mathcal{E}|) = \mathcal{O}(n)$. The identification of the desired path can also be done easily if we do some simple bookkeeping during the breadth first search. The total running time of Algorithm SMPF is thus $\mathcal{O}(nz)$ and the space required is $\mathcal{O}(nz)$. □

4.4 Experiments

Algorithm SMALLEST MAXIMAL Z-PALINDROMIC FACTORISATION was implemented as a program to compute the smallest maximal z-palindromic factorisation in one or more input sequences. The programs have been implemented in the `C++` programming language and developed under the GNU/Linux operating system. All experiments have been conducted on a Desktop PC using one core of Intel Core CPU i5-4690 at 3.50GHz. All programs have been compiled with g++ version 6.2.0 at optimisation level 3 (-O3).

In the experiment, our task was to establish the fact that the elapsed time and memory usage of the program grow linearly with n, the length of the input sequence. As input datasets, for this experiment, we used synthetic DNA sequences ranging from 250KB to 4000KB. For each sequence we used constant values for $z = 8$. The results, for elapsed time and maximal memory usage, are plotted in Fig. 4.3. It becomes evident from the results that the elapsed time and memory usage of the program grow linearly with n.



(a) Elapsed time with $z = 8$ (b) Peak memory usage with $z = 8$

Fig. 4.3 Elapsed time and peak memory usage of Algorithm Smallest Maximal z-Palindromic factorisation using synthetic DNA ($\sigma = 4$) data of length 250KB to 4000KB.

We did not apply this algorithm to real biological data since instead of palindromes, we expect more complement palindromes and gaped palindromes, which are introduced in the introduction section. Thus instead of applying the algorithm directly to real data, we need to farther development on the algorithm to make it able to work in biological research.

Chapter 5

Conclusion

5.1 Algorithms summary

In this thesis, we provide algorithms on a special sequence called Weighted Sequence. Weighted sequence is a sequence in which each position is a set of letters with occurrence probability. This sequence, which is also called as Position Weight Matrix or profile, is widely used in bioinformatics for presenting relatively sequences such as transcription factor binding sites or results from a set of aligned sequences, and become important since they can be used as an input of many biology software tools such as *Biostrings* and *seqLogo*. In this thesis, we discuss Weighted Pattern Matching Problem, Weighted Indexing problem and Palindromic factorisation Problem on weighted sequences.

In the weighted pattern matching problem, we present four sub-questions: WEIGHTEDPATTERNMATCHING, where the pattern is a weighted string and the text is a standard string; WEIGHTEDTEXTMATCHING, where the pattern is a standard string and the pattern is

a weighted string; GENERALWEIGHTEDPATTERNMATCHING, where both the pattern and text are weighted string; and APPROXWEIGHTEDTEXTMATCHING, where the pattern is a standard string and the text is a weighted string and match with k-mismatches. We provide two on-line algorithms, using partition technique, to solve problem WEIGHTEDPATTERN-MATCHINGand WEIGHTEDTEXTMATCHINGrespectively. Given a pattern of length m, a text of length n and a cumulative weight threshold $1/z \in (0,1]$, Both algorithms achieve average-case $o(n)$ search times with preprocessing time and space (m) and a weight ratio

$$\frac{z}{m} < \min\left\{\frac{1}{2\log z + \frac{1}{z}}, \frac{\log\sigma}{\log z(\log m + \log\log\sigma)}\right\}.$$

We also provide three on-line average case algorithms, using sliding window technique, to solve problem WEIGHTEDTEXTMATCHING, GENERALWEIGHTEDPATTERNMATCH-INGand APPROXWEIGHTEDTEXTMATCHING. Given a pattern of length m, a text of length n, a cumulative weighted threshold $1/z \in (0,1]$, and a integer $k > 0$ for APPROXWEIGHTED-TEXTMATCHING, the preprocessing time and searching time of our algorithms are presented in the table below, where the preprocessing time and space are worst case complexity, $0 < c < 1/2$ is an absolute constant, $v = \frac{2^\sigma - 1}{2^\sigma - 1}$, $d = 1 + (1-c)\log_v(1-c) + c\log_v c$, and $a = 4\sqrt{c(1-c)}$.

Problem	Preprocessing space	Preprocessing time	Search time	Conditions
WTM	$\mathcal{O}(m)$	$\mathcal{O}(m)$	$\mathcal{O}(\frac{nz\log m}{m})$	
GWPM	$\mathcal{O}(zm)$	$\mathcal{O}(zm)$	$\mathcal{O}(\frac{nz\log m}{m})$	
AWTM	$\mathcal{O}(\sigma^q)$	$\mathcal{O}(mq\sigma^q)$	$\mathcal{O}(\frac{nz(\log m+k)}{m})$	$q \geq \frac{3\log_\nu m-\log_\nu a}{d}$, $\frac{k}{m} \leq c-\frac{2cq}{m}$

Experimental results using synthetic DNA data are provided for all the algorithms to prove our theorems. We also provide comparing results of our algorithms against the worst-case $\mathcal{O}(nz^2\log z)$-time algorithm of [13] and the worst-case $\mathcal{O}(n\log z)$-time algorithm of [44]. The experimental results show that our algorithms are between one or two orders of magnitude faster than the best worst-case algorithms. We provide applications using real DNA sequences as well to show that our algorithms are efficient to be used in real data.

In the weighted indexing problem, we present our solutions for constructing two data structures on weighted string: weighted suffix tree and weighted suffix array. Our solutions are based on the construction of a property string called z-estimation, which contains all valid factors for a given weighted string. We prove that for any given weighted string of length n, there must exists a z-estimation of length nz and can be constructed in $\mathcal{O}(nz)$-time. We then provide our algorithm to construct property suffix tree from z-estimation, which can be transformed to weighted suffix tree directly, and show the solution with a data structure of size $\mathcal{O}(nz)$ to answer the searching queries in weighted suffix tree. In the weighted suffix array problem, we first describe how to construct a property suffix array. We provide three worse-case $\mathcal{O}(n)$-space algorithms: a Sparse-Table-based $\mathcal{O}(n\log^2 n)$-time

algorithm, a LCP-based $\mathcal{O}(nlogn)$ algorithm and a Union-Find-based $\mathcal{O}(n)$-time algorithm for the property suffix array algorithm. We also provide an average-case analysis for the Sparse-Table-based algorithm to achieve an average-case time complexity $\mathcal{O}(n)$. In the Union-Find-based algorithm, a $\mathcal{O}(n)$-time property longest common prefix array construction is provided simultaneously. Based on property suffix array, we present a modification to construct the weighted suffix array, achieving time complexity $\mathcal{O}(nz)$ for a given weighted string of length n and a cumulative weight threshold $1/z \in (0,1]$. Experimental results and comparison between the algorithms are presented using synthetic DNA data (except the LCP-based algorithm because the RMQ query is too slow in practice). The results support our time and space theorems, and the comparison between the algorithms shows us that the Union-Find-based algorithm has the best time performance and the Sparse-Table-based algorithm is the most space efficient. We also apply our algorithms on real weighted DNA sequences and show the performance results.

With the z-estimation and weighted suffix tree, we develop an algorithm, based on the algorithm by Alatabbi et al[4] on standard string, to find all maximal z-palindromes in a weighted string and the smallest maximal z-palindromic factorisation of a given weighted string. For a given weighted string of length n, our algorithm achieve time and space complexity $\mathcal{O}(n)$. Experimental results using synthetic DNA sequences are provided to prove our theoretical findings. This algorithm gives an evidence that it is available to develop some of the algorithms based on suffix tree or suffix array from standard strings to weighted strings with our weighted indexing.

5.2 Future Work

In the weighted pattern matching problem, all the algorithms presented are average-case algorithms with specific weight ratio $\frac{z}{m}$. For WEIGHTEDPATTERNMATCHING and WEIGHTEDTEXTMATCHING, the worst-case algorithms are already provided by Kociumaka et al. with time complexity of $\mathcal{O}(n \log z)$ in [44], and the worst-case algorithm for GENERALWEIGHTEDPATTERNMATCHINGover a const-size alphabet is also presented in the same paper with time complexity of $\mathcal{O}(n\sqrt{z}\log\log z)$. The worst-case algorithm for APPROXWEIGHTEDTEXTMATCHINGproblem is provided by Amir et al in [9]. However, unlike the Kociumaka's algorithms of which the time complexity is only with respect to n and z, the time complexity of Amir's algorithm is $\mathcal{O}(n\sqrt{m\log m})$, which is with respect to the pattern length m. Therefore our future work in weighted pattern matching is to develop and implement a worst-case algorithm that the time complexity is only with respect to n and z.

In the weighted indexing problem, although the time and space complexity $\mathcal{O}(nz)$ is linear with the input text length n, the space requirement in practice, observed from the experiments in Section 3.5, is still huge. One of the solutions is to build a compress property suffix tree, introduced by Hon et al. in [34], of the z-estimation of the given weighted string. To implement this index and to test the time and space performance in practice is our future work in weighted indexing.

Another future work is a multiply profile matching problem based on the weighted index. This problem comes from Zhang's research [66], and in this problem, we are given a large amounts of profiles in position weight matrix format and a DNA sequence, normally a chromosome sequence. Our aim is to find out all the profiles that occur in the given sequence,

and locate their occurrences. The algorithm of this problem is trivial, however we are aiming to show that with a built weighted suffix tree, our algorithm can have a fast enough time performance to answer the searching queries compared against other methods in practice.

References

[1] 1000 Genomes Project Consortium, Auton, A., Brooks, L. D., Durbin, R. M., Garrison, E. P., Kang, H. M. M., Korbel, J. O., Marchini, J. L., McCarthy, S., McVean, G. A., and Abecasis, G. R. (2015). A global reference for human genetic variation. *Nature*, 526(7571):68–74.

[2] Abouelhoda, M. I., Kurtz, S., and Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86.

[3] Aggarwal, C. C. and Yu, P. S. (2009). A survey of uncertain data algorithms and applications. *IEEE Trans. Knowl. Data Eng.*, 21(5):609–623.

[4] Alatabbi, A., Iliopoulos, C. S., and Rahman, M. S. (2013). Maximal palindromic factorization. In *PSC*, pages 70–77.

[5] Almirantis, Y., Charalampopoulos, P., Gao, J., Iliopoulos, C. S., Mohamed, M., Pissis, S. P., and Polychronopoulos, D. (2017). On avoided words, absent words, and their application to biological sequence analysis. *Algorithms for Molecular Biology*, 12(1):5.

[6] Alzamel, M., Charalampopoulos, P., Iliopoulos, C. S., and Pissis, S. P. (2017). How to answer a small batch of rmqs or LCA queries in practice. In Brankovic, L., Ryan, J., and Smyth, W. F., editors, *Combinatorial Algorithms - 28th International Workshop, IWOCA 2017, Newcastle, NSW, Australia, July 17-21, 2017, Revised Selected Papers*, volume 10765 of *Lecture Notes in Computer Science*, pages 343–355. Springer.

[7] Amir, A., Chencinski, E., Iliopoulos, C., Kopelowitz, T., and Zhang, H. (2008). Property matching and weighted matching. *Theoretical Computer Science*, 395(2-3):298–310.

[8] Amir, A., Gotthilf, Z., and Shalom, B. R. (2010). Weighted LCS. *J. Discrete Algorithms*, 8(3):273–281.

[9] Amir, A., Iliopoulos, C., Kapah, O., and Porat, E. (2006). Approximate matching in weighted sequences. In Lewenstein, M. and Valiente, G., editors, *Combinatorial Pattern Matching*, pages 365–376, Berlin, Heidelberg. Springer Berlin Heidelberg.

[10] Apostolico, A., Breslauer, D., and Galil, Z. (1995). Parallel detection of all palindromes in a string. *Theoretical Computer Science*, 141(1):163–173.

[11] Barton, C., Iliopoulos, C. S., and Pissis, S. P. (2014). Optimal computation of all tandem repeats in a weighted sequence. *Algorithms for Molecular Biology*, 9(21).

[12] Barton, C., Kociumaka, T., Pissis, S. P., and Radoszewski, J. (2016). Efficient index for weighted sequences. In *CPM*, volume 54 of *LIPIcs*, pages 4:1–4:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[13] Barton, C. and Pissis, S. P. (2015). Linear-time computation of prefix table for weighted strings. In *WORDS 2015*, volume 9304 of *LNCS*, pages 73–84. Springer.

[14] Barton, C. and Pissis, S. P. (2017). Crochemore's partitioning on weighted strings and applications. *Algorithmica*.

[15] Bender, M. A. and Farach-Colton, M. (2000). The LCA problem revisited. In *LATIN*, volume 1776 of *LNCS*, pages 88–94. Springer-Verlag.

[16] Birney, E. and Soranzo, N. (2015). The end of the start for population sequencing. *Nature*, 526:52 EP –.

[17] Biswas, S., Patil, M., Thankachan, S. V., and Shah, R. (2016). Probabilistic threshold indexing for uncertain strings. In *EDBT*, pages 401–412. OpenProceedings.org.

[18] Caspi, R., Helinski, D. R., Pacek, M., and Konieczny, I. (2000). Interactions of DNAA proteins from distantly related bacteria with the replication origin of the broad host range plasmid RK2. *The Journal of Biological Chemistry*, 275(24):18454–18461.

[19] Chang, W. I. and Marr, T. G. (1994). Approximate string matching and local similarity. In *CPM 1994*, CPM '94, pages 259–273. Springer-Verlag.

[20] Christodoulakis, M., Iliopoulos, C. S., Mouchard, L., Perdikuri, K., Tsakalidis, A. K., and Tsichlas, K. (2006). Computation of repetitions and regularities of biologically weighted sequences. *Journal of Computational Biology*, 13(6):1214–1231.

[21] Christodoulakis, M., Iliopoulos, C. S., Mouchard, L., and Tsakalidis, A. K. (2004). Pattern matching on weighted sequences. In *CompBioNets*, KCL publications.

[22] Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. (2001). *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition.

[23] Crochemore, M., Czumaj, A., Gasieniec, L., Jarominek, S., Lecroq, T., Plandowski, W., and Rytter, W. (1994). Speeding up two string-matching algorithms. *Algorithmica*, 12(4/5):247–267.

[24] Crochemore, M., Czumaj, A., Gasieniec, L., Lecroq, T., Plandowski, W., and Rytter, W. (1999). Fast practical multi-pattern matching. *Inf. Process. Lett.*, 71(3-4):107–113.

[25] Crochemore, M., Iliopoulos, C., Kubica, M., Radoszewski, J., Rytter, W., Stencel, K., and Walen, T. (2014). New simple efficient algorithms computing powers and runs in strings. *Discrete Applied Mathematics*, 163(Part 3):258–267.

[26] Cygan, M., Kubica, M., Radoszewski, J., Rytter, W., and Walen, T. (2011). Polynomial-time approximation algorithms for weighted LCS problem. In *CPM*, volume 6661 of *LNCS*, pages 455–466. Springer.

[27] Farach, M. (1997). Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 137–143.

[28] Fici, G., Gagie, T., Kärkkäinen, J., and Kempa, D. (2014). A subquadratic algorithm for minimum palindromic factorization. *Journal of Discrete Algorithms*, 28:41–48.

[29] Franklin, R. E. and Gosling, R. G. (2004). Molecular configuration in sodium thymonucleate. *Resonance*, 9(3):84–88.

[30] Gabow, H. N. and Tarjan, R. E. (1985). A linear-time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci.*, 30(2):209–221.

[31] Guo, Y. and Jamison, D. C. (2005). The distribution of SNPs in human gene regulatory regions. *BMC Genomics*, 6(1):1–11.

[32] Gusfield, D. (1997). *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, NY, USA.

[33] Hattori, M. et al. (2000). The DNA sequence of human chromosome 21. *Nature*, 405:311–319.

[34] Hon, W.-K., Patil, M., Shah, R., and Thankachan, S. V. (2013). Compressed property suffix trees. *Information and Computation*, 232:10 – 18.

[35] Hui, L. C. K. (1992). Color set size problem with application to string matching. In *Combinatorial Pattern Matching, Third Annual Symposium, CPM 92, Tucson, Arizona, USA, April 29 - May 1, 1992, Proceedings*, pages 230–243.

[36] I, T., Sugimoto, S., Inenaga, S., Bannai, H., and Takeda, M. (2014). Computing palindromic factorizations and palindromic covers on-line. In *CPM*, volume 8486 of *LNCS*, pages 150–161. Springer International Publishing.

[37] Iliopoulos, C. S., Makris, C., Panagis, Y., Perdikuri, K., Theodoridis, E., and Tsakalidis, A. K. (2006). The weighted suffix tree: An efficient data structure for handling molecular weighted sequences and its applications. *Fundam. Inform.*, 71(2-3):259–277.

[38] Iliopoulos, C. S., Mouchard, L., Perdikuri, K., and Tsakalidis, A. K. (2005). Computing the repetitions in a biological weighted sequence. *Journal of Automata, Languages and Combinatorics*, 10(5/6):687–696.

[39] Iliopoulos, C. S., Perdikuri, K., Theodoridis, E., Tsakalidis, A. K., and Tsichlas, K. (2004). Motif extraction from weighted sequences. In *SPIRE*, volume 3246 of *LNCS*, pages 286–297. Springer.

[40] Iliopoulos, C. S. and Rahman, M. S. (2008). Faster index for property matching. *Inf. Process. Lett.*, 105(6):218–223.

[41] Juan, M. T., Liu, J. J., and Wang, Y. L. (2009). Errata for "faster index for property matching". *Inf. Process. Lett.*, 109(18):1027–1029.

[42] Karlin, S., Ghandour, G., amd Simon Tavare, F. O., and Korn, L. J. (1983). New approaches for computer analysis of nucleic acid sequences. *Proceedings of the National Academy of Sciences of the United States of America*, 80(18):5660–5664.

[43] Kasai, T., Lee, G., Arimura, H., Arikawa, S., and Park, K. (2001). Linear-time longest-common-prefix computation in suffix arrays and its applications. In *CPM*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer.

[44] Kociumaka, T., Pissis, S. P., and Radoszewski, J. (2016). Pattern matching and consensus problems on weighted sequences and profiles. In *ISAAC 2016*, LIPIcs. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[45] Kociumaka, T., Pissis, S. P., Radoszewski, J., Rytter, W., and Walen, T. (2018). Efficient algorithms for shortest partial seeds in words. *Theor. Comput. Sci.*, 710:139–147.

[46] Kopelowitz, T. (2016). The property suffix tree with dynamic properties. *Theor. Comput. Sci.*, 638:44–51.

[47] Lovász, L., Pelikán, J., and Vesztergombi, K. (2003). *Discrete Mathematics: Elementary and Beyond*. Springer, New York, NY, USA.

[48] Manacher, G. (1975). A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. *Journal of the ACM*, 22(3):346–351.

[49] Manber, U. and Myers, E. W. (1993). Suffix arrays: A new method for on-line string searches. *SIAM Journal of Computing*, 22(5):935–948.

[50] Maxam, A. M. and Gilbert, W. (1977). A new method for sequencing dna. *Proceedings of the National Academy of Sciences*, 74(2):560–564.

[51] Muhire, B. M., Golden, M., Murrell, B., Lefeuvre, P., Lett, J.-M., Gray, A., Poon, A. Y., Ngandu, N. K., Semegni, Y., Tanov, E. P., et al. (2014). Evidence of pervasive biologically functional secondary structures within the genomes of eukaryotic single-stranded DNA viruses. *Journal of virology*, 88(4):1972–1989.

[52] Musser, D. R. (1997). Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8):983–993.

[53] Muthukrishnan, S. (2002). Efficient algorithms for document retrieval problems. In Eppstein, D., editor, *13th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2002*, pages 657–666. ACM/SIAM.

[54] Navarro, G. (2001). A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88.

[55] Navarro, G. and Nekrich, Y. (2012). Top-k document retrieval in optimal time and linear space. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 1066–1077.

[56] Nong, G., Zhang, S., and Chan, W. H. (2009). Linear suffix array construction by almost pure induced-sorting. In *DCC*, IEEE, pages 193–202.

[57] Radoszewski, J. and Starikovskaya, T. A. (2017). Streaming k-mismatch with error correcting and applications. In *DCC*, pages 290–299. IEEE.

[58] Rubinchik, M. and Shur, A. M. (2016). Eertree: An efficient data structure for processing palindromes in strings. In *IWOCA*, volume 9538 of *LNCS*, pages 321–333. Springer International Publishing.

[59] Sandelin, A., Alkema, W., Engström, P., Wasserman, W. W., and Lenhard, B. (2004). JASPAR: an open-access database for eukaryotic transcription factor binding profiles. *Nucl. Acids Res.*, 32(1):D91–D94.

[60] Sanger, F. and Coulson, A. R. (1975). A rapid method for determining sequences in dna by primed synthesis with dna polymerase. *J Mol Biol*, 94(3):441–448.

[61] Stormo, G. D., Schneider, T. D., Gold, L., and Ehrenfeucht, A. (1982). Use of the 'perceptron' algorithm to distinguish translational initiation sites in e. coli. *Nucleic Acids Research*, 10(9):2997–3011.

[62] ten Bosch, J. R. and Grody, W. W. (2008). Keeping up with the next generation: Massively parallel sequencing in clinical diagnostics. *The Journal of Molecular Diagnostics*, 10(6):484 – 492.

[63] Ukkonen, E. (1995). On-line construction of suffix trees. *Algorithmica*, 14(3):249–260.

[64] Watson, J. D. and Crick, F. H. (1953). Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid. *Nature*, 171:737–738.

[65] Yao, A. C.-C. (1979). The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387.

[66] Zhang, Z., Chang, C. W., Goh, W. L., Sung, W.-K., and Cheung, E. (2011). Centdist: discovery of co-associated factors by motif distribution. *Nucleic Acids Research*, 39(suppl_2):W391–W399.